

تعلم كل شيء عن

JAVA

PROGRAMMING FOR INTERNET

online



JAVA

Programming Lang

```

VAR IsBrowserOk="NO"
lok="home"
link= new Array ("home", "site", "free")
function GetButtonOn (page, arrow)
{
    if (IsBrowserOk=="yes")
    {
        if (page !=lok)
        {
            document[arrow] src="squre.gif"
        }
    }
}
    
```



CONTACT WITH the WORLD

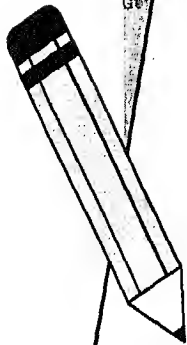
المهندس : مهيب النكري

تعلم كل شيء عن

Java

إعداد

المهندس مهيب النقري



بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

٣٥

سلسلة الرضا للمعلومات

سلسلة علمية متميزة لنشر ثقافة الإدارة الحديثة والمعلوماتية
لتطوير المؤسسات والشركات التي تسعى للريادة.

المراجعة العلمية : د. صلاح الدوه جي

م. سامر سعيد

م. حسن شاليش حسن

التدقيق اللغوي : لميس فرحة

مركز الرضا للكمبيوتر - دار الرضا للنشر

تجهيز - قرب فندق برج الفردوس

هاتف: ٢٢٢٤٦١٧ - تليفاكس: ٢٢٢٢١٦٣ - ص.ب: ٤٢٦٧

حقوق النشر محفوظة

أيلول ١٩٩٩

الإهداء

إلى أغلى مالدي في الوجود
إلى ابنتي ميرنا

لابد من كلمة شكر...

أشكر كل من ساهم في إنجاز هذا العمل بشكله الحالي، كل من أبدى ولو برأي بسيط، وحتى بفكرة...
شكري العميق لكل من ساهم في المراجعة العلمية لهذا الكتاب، وأخص بالشكر الدكتور صلاح الدوه جي، والمهندس سامر سعيد، والمهندس حسن شاليش حسن.
شكري العميق لمن كانت صديقتي ومساعدتي على إنجاز هذا العمل، وقامت بتدقيقه وإهداء الملاحظات القيمة، شكري إلى زوجتي.
كل الشكر للصديق الدكتور هاني الخوري مدير مركز الرضا للكمبيوتر على كل مساعدة أبداهما لإتمام هذا العمل.

تقديم الناشر

تدور حركة التطور البشري العالمية اليوم بكل أبعادها ومحاورها الاقتصادية والعلمية والثقافية والاجتماعية في إطار تكنولوجيا المعلومات، وارتباطها بتطور الاتصالات وعلوم الإدارة العلمية الحديثة، وهو محور شامل غير طبيعية العلاقات الاقتصادية والبشرية والثقافية والعلمية وجعلها تدور في ديناميكية وتغير متسارع، وطرح تحديات كبيرة لكل المجتمعات للانتقال بقوة ونجاح إلى معطيات القرن الحادي والعشرين.

لا أبالغ إذا قلت أن مختلف الشعوب اليوم تعيش حالة تحضير شامل على كل الصعد لتحديث البنى والأفكار والتوجهات وبناء الكوادر البشرية وتنظيم المؤسسات من جديد، بناءً على التطور المذهل في تكنولوجيا ونظم المعلومات، وما تبعها من تطور في وسائل الاتصال والإعلام العالمي من فضائيات ومن شبكة عالمية، اكتنفت علوم الأرض العالمية كلها بكل ديناميكية وشمولية بعد أن لفت خيوط عناكبها بلدان العالم أجمع لتبني مفهوم القرية الكونية GLOBAL VILLAGE.

في إطار هذا المحور نمت وتوطدت وتوسعت أبعاد ظاهرة العولمة مدفوعة بزخم القطبية الأحادية والتفوق التكنولوجي الأمريكي، وقد تزامنت هذه التحديات مع استحقاقات أصعب هي استحقاقات الانفتاح التجاري العالمي واتفاقية الجات.

في خضم هذه التحديات تفرض اليوم على كل المجتمعات والدول استحقاقات إعادة تحديث تكنولوجي وإداري شامل، لبناء المجتمع ومؤسساته وبناءه الاقتصادية والعلمية والثقافية بأسس تصلح لمواجهة تحديات القرن الحادي والعشرين، وأنا أؤكد هنا على أولوية وأهمية واستراتيجية تطوير العنصر البشري والكفاءات البشرية، بدءاً بتعديل أساليب ومنهجيات التعليم من التلقين إلى المشاركة والمواكبة والتحديث إلى بناء الكوادر المؤسسية ودعمها بالتدريب والتأهيل الدائم، للتغلب على تحديات التضاعف المعرفي السريع والتحدي التقني وتسارع التغييرات، فالنقط والثروات تنضب أما الخيار البشري والعقل فهما مصدر الغنى الدائم، ولا أرى هنا مثلاً أشد دلالة من اليابان.

لقد تطورت التحديات المعلوماتية وامتزجت بالتطور الاتصالي ولا سيما الانترنت التي أصبحت بنك المعلومات الحضاري الشامل وأصبح عدد مستخدميها يعد بمئات الملايين لتهيئة العالم نحو مفهوم جديد بالتواصل والثقافة والعلم والاقتصاد والتجارة وفتحت الآفاق نحو مفهوم الاقتصاد المعرفي وتبادل السلع المعرفية المعلوماتية من تصاميم وبرامج وأفكار وأفلام وحلول وإحصائيات ومعلومات ولتخلق مفهوم الكوادر الكونية ومفهوم العمل عن بعد في المنازل وعبر الدول والحدود لدمج الكفاءات العالمية لصالح الاقتصاديات والشركات الكبرى المتعددة الجنسيات إن تحديات ذلك الاقتصاد المعرفي والتجارة عبر الانترنت تفتح آفاق تحديث البنية التحتية الاتصالية والقانونية والتبادل المالي عبر الانترنت من خلال المصارف المؤتمنة وتحديث المفاهيم الثقافية والعملية باتجاه أعمال جديدة تؤكد على الانتاج الفكري والسلع القابلة للتبادل عبر الانترنت.

وهنا تأتي لغة جافا كلفة علمية حديثة حدثت خصيصاً من أجل تطبيقات الانترنت الغير معتمدة على نظم التشغيل لتفتح المجال لجيل جديد من اللغات ونوعية البرمجة التي تهتم باحتياجات الانترنت وتطبيقاتها ومواقعها وخدماتها، وهذا ما قام به المهندس مهيب النقري بتقديم كتابه تعلم كل شيء عن لغة جافا، الذي يمتاز بموضوعات جديدة ترتبط بقواعد البيانات وبنظام أوراكل وغيرها من المواضيع التي للمرجع أهميته.

إن اهتمام مركزنا مستمر بإصدار العديد من المراجع حول العلوم والتطبيقات الحديثة التي تهتم بالانترنت لما لهذا المجال الحديث من ضرورات بالتعريف والتطوير، ونتمنى أن يقدم هذا المرجع والكتب القادمة عن انترنت الفائدة العلمية المرجوة لكل قارئ بمشيئة الله. والله من وراء القصد

دمشق في ١٩٩٩/٩/٩

مدير دار الرضا للنشر

هاني شحادة الخوري



١. المقدمة ... ١٧

٢. المفاهيم الأساسية للبرمجة غرضية التوجه *Object Oriented* ... ٢٥

٢٥ ... *Programming Concepts*

٢٧ ... *Abstraction* التجريد

٢٨ ... *Interface* لكل عنصر واجهة إظهار

٢٩ ... *The Hidden Implementation* التنفيذ المخفي

٣٠ ... *Reusing the implementation* إعادة استخدام الترميز

٣١ ... *Inheritance* التوريث

٣٣ ... *Polymorphism* تعددية الأشكال

- ٣٥ ...Abstract Base Classes الصفوف الأساسية المجردة
- ٣٦ ...Object Landscapes and Lifetimes مناظر وأعمار العناصر
- ٣٧ ...Collections and Iterators المجموعات والتكرارات
- ٣٩ ...Exception Handling معالجة الاستثناءات
- ٣٩ ...Multithreading تعددية النياسب

٣. لنبدأ بالتعرف على لغة جافا... ٤١

- ٤٢ يمكنك التعامل مع العناصر باستخدام المؤشرات...
- ٤٣ يجب عليك إنشاء جميع العناصر...
- ٤٣ لكن أين يتم تخزين البيانات؟
- ٤٥ الأنماط الأولية...
- ٤٧ المصفوفات في جافا...
- ٤٧ لن تكون بحاجة أبداً لتدمير عنصر...
- ٤٨ لتتعرف أولاً على مفهوم نطاق العمل...
- ٤٩ لتتعرف الآن على نطاق عمل العناصر...
- ٤٩ إنشاء الصفوف...
- ٥٠ بناء الحقول...
- ٥٢ الطرق Method...
- ٥٣ لنبدأ إذاً بإنشاء أول برنامج جافا...
- ٥٣ معايير التسميات في جافا...
- ٥٤ وإذا احتجت لاستخدام مكونات أخرى، ماذا أفعل؟
- ٥٥ وماذا تعني كلمة المفتاح static؟
- ٥٧ ساعلمك إذا كتابة أول برنامج بلغة جافا...
- ٥٨ تساعدك لغة جافا حتى على توليد توثيق لبرامجك...
- ٥٨ لتتعرف الآن على تركيبة التوثيق...



- ما الذي تعنيه بـ *HTML* المضمنة؟ ٥٩
 تفيدك علامة *@see* للدلالة على صفوف أخرى... ٦٠
 علامات توثيق الصف... ٦٠
 علامات توثيق المتحولات... ٦٠
 علامات توثيق الطرق... ٦١

٤. معاملات وتعليمات لغة جافا... ٦٣

- المعاملات في جافا... ٦٤
 وماهي أفضليّات المعاملات في لغة جافا؟ ٧٠
 التعليمات الأساسية في جافا... ٧٠
 تعليمة الشرط *if-else*... ٧٠
 تعليمات التكرار في جافا؟ ٧٢
 حتى أنه يمكنك استخدام تعليمة *goto* المتخلفة؟! ٧٤
 تعليمة الاختيار *Switch*؟ ٧٦

٥. حلّت جافا جميع المشاكل المتعلقة بالقيمة الابتدائية ومسح العناصر... ٧٩

- عند استخدامك للبيانات *Constructors* ستتخلص من مشكلة تحديد القيمة الابتدائية... ٨٠

تحميل الطرق *Method Overloading* ... ٨٢

- لكن كيف تستطيع جافا التمييز بين الطرق المحملة *Overloaded Method*؟ ٨٤
 التحميل باستخدام الأنماط الأولية... ٨٤
 لقد نسيت إنشاء بانٍ ضمن الصف، فماذا أفعل؟ ٨٧
 ما هي الفائدة من كلمة المفتاح *this*؟ ٨٧
 يمكنك أيضاً استدعاء بانٍ من خلال بانٍ آخر؟! ٨٩

مجمع النفايات *Garbage Collector* ... ٩٠

- لكن كيف يمكن إنجاز عملية المسح؟ ٩١

تحديد القيم الابتدائية لعضو... ٩٥

كيف تقوم جافا بتحديد القيم الابتدائية للمتحوّلات الساكنة؟ ٩٧

وماذا عن المتحوّلات غير الساكنة؟ ٩٧

تحديد القيم الابتدائية للمصفوفات... ٩٨

٦. استخدام المكتبات والصفوف *Using Libraries and Classes* ١٠٥التعامل مع الحزم *Packages*... ١٠٦محدّدات الوصول *Access Specifiers*... ١٠٨المحدّد الصديق *Friendly*... ١٠٩المحدّد العام *Public*... ١٠٩المحدّد الخاص *Private*... ١١١النمط المحمي *Protected*... ١١١

لكن ماذا بالنسبة إلى تحديد سماحية الوصول إلى الصفوف؟ ١١٣

الصفوف *Classes*... ١١٤التركيب *Composition*... ١١٤التوريث *Inheritance*... ١١٨أصبح أخيراً هناك معنى لاستخدام محدّد الوصول *protected*... ١٢٣أصبح أيضاً هناك معنى للتحميل للأعلى *Upcasting*... ١٢٤*Final* أخيراً وليس آخراً... ١٢٥المعطيات النهائية *Final Data*... ١٢٥الطرق النهائية *Final Methods*... ١٢٧الصفوف النهائية *Final Classes*... ١٢٧٧. تعدّدية الأشكال *Polymorphism*... ١٢٩التوجيه للأعلى *Upcasting*... ١٣٠

لكن ما الفائدة من التوجيه للأعلى؟ ١٣١



الربط *Binding* ؟ ١٣٤الصفوف والطرق المجردة *Abstract Classes and Methods* ... ١٣٥

يوجد دور للصفوف والطرق المجردة في بناء الواجهة... ١٣٩

التوريث المتعدد *Multiple Inheritance* ... ١٤٣

يمكنك توسيع واجهتك باستخدام التوريث... ١٤٥

بإمكانك أيضا استخدام الواجهات لإنشاء مجموعات من الثوابت... ١٤٦

الصفوف الداخلية *Inner classes* ... ١٤٦

البيانات وتعددية الأشكال... ١٥٠

التوريث والطريقة *finalize()* ... ١٥١

٨. وأين بإمكانني وضع عناصره؟ ١٥٥

المصفوفات *Arrays* ... ١٥٦

هل تسمح لي جافا بإرجاع مصفوفة؟! ١٥٩

المجموعات *Collections* ؟ ١٦١

لكن انتبه، فأنت ستتعامل مع نمط غير معروف!! ١٦١

هناك نوع من العدادات اسمه *Enumeration* ... ١٦٢المجموعة الأبسط *Vector* ... ١٦٤يوجد نمط آخر قريب من *Vector* هو *BitSet* ... ١٦٤المكدس *Stack* ... ١٦٦النمط الرابع والأخير هو *hashtable* ... ١٦٧

أصبح هناك مكتبة مجموعات شاملة... ١٦٩

كيفية التعامل مع المجموعات *Collection* ... ١٧٠القوائم *List* ؟ ١٧٣وما الجديد في المجموعات *Set* ؟ ١٧٨لنتحدث أخيرا عن الطابق *Map* ... ١٧٩

٩. معالجة الأخطاء باستخدام الاستثناءات ... ١٨٣

لنتحدث بتفصيل أكثر عن الاستثناءات ... ١٨٤

كيف يتم إذا النقاط استثناء؟ ١٨٥

يمكنك التقاط استثناء باستخدام كلمة *try* ... ١٨٦

هناك طريقة إجبارية لتوصيف استثناء ... ١٨٧

وكيف أستطيع التقاط أي استثناء؟ ١٨٨

ماهي الاستثناءات القياسية في لغة جافا؟ ١٨٩

يمكنك أيضا إنشاء استثناءاتك الخاصة ... ١٩٠

١٠. نظام الإدخال والإخراج في جافا ... ١٩٣

أنماط الصف *Reader* ... ١٩٤

أنماط الصف *Writer* ... ١٩٥

ماهي الفائدة من استخدام الصف *File*؟ ١٩٩

بإمكانك تقسيم نص باستخدام الصف *StreamTokenizer* ... ٢٠٣

يمكنك القيام بنفس العمل باستخدام الصف *StringTokenizer* ... ٢٠٧

إعادة توجيه الدخل والخرج القياسي ... ٢٠٩

بإمكانك ضغط بياناتك أيضا؟ ٢١١

هناك أيضا أداة ممتازة للأرشفة ... ٢١٤

سلسلة العناصر *Object serialization* ... ٢١٤



١١. إنشاء البرمجيات والنوافذ *Creating applets and windows*

٢١٩ ...*windows*

٢٢٠ ...*The Basic Applet* البرمج الأساسي

٢٢٣ ...*Creating a Button* إنشاء زر

٢٢٤ ...*Capturing an Event* التقاط حدث

٢٢٥ ...*Text Fields* الحقول النصية

٢٢٧ ...*Text Areas* المناطق النصية

٢٢٨ ...*Labels* اللصاقات

٢٣٠ ...*Check Boxes* صناديق التحقق

٢٣٢ ...*Radio Buttons* أزرار الراديو

٢٣٣ ...*Drop-Down Menus* اللوائح المتدلية

٢٣٥ ...*List Boxes* صناديق اللائحة

٢٣٧ ...*Controlling layout* التحكم بالتخطيط

٢٣٧ ...*FlowLayout*

٢٣٨ ...*BorderLayout*

٢٣٩ ...*GridLayout*

٢٣٩ ...*CardLayout*

٢٤٠ ...*action()* بدائل الطريقة

٢٤٦ ...*Windowed Applications* إنشاء نوافذ التطبيقات

٢٤٦ ...*Menus* القوائم

٢٥٠ ...*Dialog Boxes* صناديق الحوار

- مكتبة AWT في الإصدار Java 1.1 ... ٢٥٥
 نموذج الحدث الجديد New Event Model ... ٢٥٥
 الحدث وأنماط المستمع Event and listener types ... ٢٥٧

١٢. جافا والبرمجة المرئية Java and Visual Programming ٢٧١

- حبيبات جافا Java Beans ... ٢٧٢
 الحصول على BeanInfo باستخدام Introspector ... ٢٧٥
 سأعطيك الآن مثالا مسليا أكثر ... ٢٨٢
 تخزين الحبيبات Packaging Beans ... ٢٨٦

١٣. تجزيء البرامج إلى مهام جزئية باستخدام MultiThreading ٢٨٧

- يمكنك إنشاء واجهات مستخدم سريعة الاستجابة ... ٢٨٨
 ساعدني إذا على حل هذه المشكلة ... ٢٩١
 ويمكنك مشاركة المصادر المقيدة ... ٢٩٤
 ما هي حالات النيسب؟ ٣٠٠
 لكن ما هي الأسباب التي تجعلنا نقوم بتجميد نيسب؟ ٣٠١
 توجد أفضليات للنياسب ... ٣١٣
 مجموعات النياسب ... ٣١٤

١٤. جافا وبرمجة الشبكات Java and Network Programming ٣١٧

- عليك أولا تعريف جهازك ... ٣١٨
 المقابس Sockets ... ٣٢٠
 التعامل مع مخدّم/زبون بسيط ... ٣٢١
 تخديم عدة زبائن في نفس الوقت !!! ٣٢٦



استخدام البروتوكول *UDP* ... ٣٢٩

١٥. جافا وتطبيقات الويب *Web* ... ٣٣٥

سنبدأ أولاً بإنشاء تطبيق المخدم ... ٣٣٧

سنقوم بعد ذلك بإنشاء الريمج *NameSender* ... ٣٤٣

وماذا عن صفحة الوب؟ ٣٤٨

١٦. جافا وقواعد المعطيات ... ٣٥١

أداة الربط مع قواعد المعطيات *JDBC* ... ٣٥٢

أفضل توضيح ذلك بمثال عملي ... ٣٥٥

سنقوم بتوليد نسخة بواجهة مستخدم رسومية *GUI* لبرنامج البحث السابق ... ٣٥٩

جافا وقواعد معطيات أوراكل ... ٣٦٢

تطوير تطبيقات المخدم/الزبون *Client/Server Application Development* ... ٣٦٣

تطوير التطبيقات متعددة الطبقات *Multi-Tier Application Development* ... ٣٦٥

تطبيقات *JWeb* ... ٣٦٨

النفاد إلى إجراءات قاعدة المعطيات المخزنة *Accessing Database Stored*

Procedures ... ٣٦٨

تطبيقات *JCORBA* ... ٣٦٨

١٧. سنتعرف على مكتبة *Swing* ... ٣٧١

يمكنك قلب برامجك القديمة بسهولة ... ٣٧٢

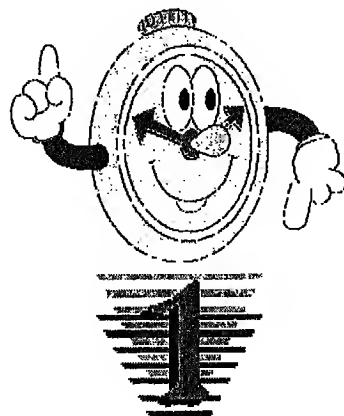
إظهار إطار عمل *framework* ... ٣٧٤

صناديق إيضاح الأدوات *Tool tips* ... ٣٧٥

الإطارات *Borders* ... ٣٧٥

الأزرار ...Buttons	٣٧٧
مجموعات الأزرار ...Button Groups	٣٧٩
الأيقونات ...Icons	٣٨١
القوائم ...Menus	٣٨٣
القوائم المنبثقة ...Popup Menus	٣٩٠
صناديق القائمة وصناديق السرد والتحرير List boxes and combo boxes	٣٩٢
أشرطة التقدم ...Sliders and progress bars	٣٩٣
الأشجار ...Trees	٣٩٤
الجدول ...Tables	٣٩٨
اللوحات المبوبة ...Tabbed Panes	٤٠٠
المراجع ...	٤٠٣
المصطلحات ...	٤٠٥





ظهرت لغة جافا عام ١٩٩٥، حيث قدّمتها شركة Sun من خلال مجموعة أدوات *Stone Age Unix*. ولم يمض وقت طويل حتى ظهرت في السوق أدوات لتطوير جافا من مستوى رفيع، حيث استجابت شركة مايكروسوفت للظاهرة بتوفير نسخة *J++* والتي دمجت جافا في *ActiveX* بشكل متكامل. كما قامت شركة بورلاند بتقديم *Java Builder* وهي بيئة لتطوير جافا

بحيث تتيح توليد ترميز لا يتداخل مع ترميز أخرى أو ما يسمى بـ *Pure Java* كما تنتج حبيبات جافا *Java Beans* والتي يتولد عنها عناصر متينة تستطيع تطويرها حسب الحاجة و معالجتها بسهولة، وتبني واجهات المستخدم الرسومية.

ولقد وضعت شركة *Sun* في هذه اللغة خصائص مميزة جداً، فمن خلال هذه اللغة، يمكن للمستخدم طلب التطبيقات عبر الإنترنت ومن ثم تشغيلها على حاسوب محلي. ولا يحتاج المرسل إلى معرفة شكل بيئة المستخدم، إن كان من جهة التجهيزات أو البرمجيات. كما أنها تجعل من انتقال الفيروسات أمراً مستحيلاً.

وكلفة فإن جافا تشبه نسخة مبسطة من لغة *C++*، مما يقلل من وقت تعلمها لدى المطورين. وأكثر من ذلك فإنها الأكثر أماناً على شبكة الإنترنت.

ولأن جافا لا تعتمد على نظام التشغيل، فإن كل ما يحتاجه مطورو البرمجيات هو إنشاء نسخة من تطبيقاتهم، وليس هناك حاجة لاختبار نسخ إضافية يعمل كل منها مع نظام تشغيل معين مثل *Windows* أو *Macintosh* أو *Unix* أو غيرها.

أما بالنسبة لمديري الأنظمة ومسؤولي تقنية المعلومات، فإن جافا تسهل عملية التحكم بالمراجعة والدخول، إذ أنها تتطلب نسخة واحدة فقط في موقع تحكم واحد. ويتم تحميل هذه النسخة للمستخدم عند التشغيل.

ويدعى الترميز الناتج عن جافا بالبرمج *Applet*، وهو يشحن من خلال الشبكة. ويمكن تشبيه هذه البرامج الصغيرة بقطع حجرية ترص معاً لبناء البيت الذي هو في الواقع التطبيق المكتوب بلغة جافا.

ومن الميزات الخاصة في جافا أنها لا تسمح بإنشاء مؤشرات خارج ترميزها الخاص. إذ أن المؤشرات الخارجية التي لا تسمح بها جافا هي التي تساعد على عمل الفيروسات المنتشرة حالياً إذ أن الضرر يحدث عندما يخرج الترميز عن مساحة الذاكرة الخاصة به، لذلك فإن برامج جافا لا تنقل الفيروسات.

كما أن جافا تدعم عالم الشبكات بشكل كبير جداً، إذ أن واجهات بروتوكولات الإنترنت مثل بروتوكول *TCP/IP* وبروتوكول *HTTP* موجودة في بنية هذه اللغة.



لذلك تعتبر لغة جافا من أقوى وأحدث لغات البرمجة التي بنيت للعمل على جميع الأنظمة ومنصات العمل *Platforms*. ولقد صممت هذه اللغة لحل الكثير من مشاكل البرمجة المتعلقة بجهة الزبون *Client* في بيئة *Client/Server*. ويمكنها أيضاً حل جميع المشاكل التي تصعب على لغات البرمجة التقليدية كتعدد النيايب *Database Access* والوصول إلى قاعدة المعطيات *Database* برمجة الشبكة *Network Programming* والبرمجة الموزعة *Distributed Programming*.

وكما ذكرنا سابقاً فإن هذه اللغة تستطيع برمجة ما يتعلق بموقع الزبون *Client-Side Programming* باستخدام ما يسمى بالبرمجيات *Applet*. و البرمج *Applet* عبارة عن برامج صغيرة يمكنها العمل من خلال مستعرضات الويب *Web Browsers* فقط، ويتم تحميلها كجزء من صفحة الويب، وعند تفعيلها يتم تنفيذ البرنامج الموافق. وتكون عادةً بشكل مترجم *Compiled Form* لذلك فإن الترميز المصدر لن يكون متاحاً للزبون. وتمتلك لغة جافا إمكانية حل المشاكل العديدة على شبكة وب العالمية *World Wide Web*.

والحق يقال بأنه لم تحظ لغة برمجة جديدة في تاريخ الكمبيوتر بدعم صنّاع الأدوات البرمجية ومطوري التطبيقات ومصنعي أنظمة التشغيل في وقت قصير مثلما حظيت به لغة جافا.

ولقد ارتقت جافا من كونها مجرد لغة حتى أصبحت بيئة للتطبيقات *Application Environment*. ويعود الفضل في ذلك إلى آلة جافا الافتراضية *Java Virtual Machine* والتي تحاكي برمجياً عمل الكمبيوتر.

وكما ذكرنا فإن لغة جافا تشبه لغة *C++*، إلا أنها تتفوق عليها في كثير من الأمور، فهي تتيح بناء ترميز متنقل وقابل لإعادة الاستخدام وخال من العثرات *Bug-free*. كما أن ميزة "اكتب مرة، وشغل أينما تريد" تكسب جافا طابعاً كلياً ملفتاً، إلى حد أن بعض

الشركات تقوم بكتابة أدوات برمجة للغة جافا باستخدام اللغة ذاتها لكي تعمل على أي جهاز.

وللمرة الأولى يمكن للمطورين كتابة برامجهم على مختلف أنظمة التشغيل *Windows* أو *Unix* أو *Mac/Os* أو غيرها. فجميع المبرمجون يمكنهم العمل على المشروع نفسه بالأدوات نفسها و في أي بيئة. وهذا يسمح بتوظيف المبرمجين استناداً لخبراتهم في برمجة التطبيق الذي يريدون، وليس لمعرفتهم بالنظام الذي يستخدمونه.

ونظراً لأن تطبيقات جافا تبقى داخل بيئة جافا للتشغيل، فإنها لا تتفاعل مباشرة، مع وحدة المعالجة المركزية أو نظام التشغيل. فبيئة تشغيل جافا تعالج مسائل الذاكرة ذاتياً، بحيث لا يحتاج المبرمج للقيام بتخصيص الذاكرة، أو تفريغ الترميز منها. كذلك لا حاجة لاستخدام المؤشر الحسابي *Pointer* والذي يعدّ أحد مصادر الأخطاء والعثرات في لغة *C++*. وضمن لغة جافا هناك نموذج فعال في معالجة الأخطاء يشجّع على إعادة استخدام الترميز كونها بنيت لاستخدام العناصر أصلاً. وتستبدل جافا التوريث المعقّد المتعدّد السجود في لغة *C++* بالواجهات *Interfaces*.

سنحاول في هذا الكتاب التعريف بهذه اللغة بشكل مفصل وذلك عن طريق شرح الميزات الأساسية الموجودة فيها مع إعطاء الأمثلة المناسبة. ولقد اعتمدنا هنا على استخدام لغة برمجة جافا القياسية بمختلف إصداراتها الموجودة، دون محاولة استخدام أي من أدوات ومترجمات جافا الموجودة في السوق مثل *Microsoft Visual J++* أو *Java Builder* أو غيرها. وتركنا ذلك للكتاب القادم الذي سيصدر عن الدار وهو بعنوان "جافا والإنترنت" والذي سيتم فيه أخذ أحد أهم الجوانب التطبيقية للغة جافا. وهدفنا من ذلك إتاحة المجال للقارئ لفهم تعليمات وأفكار لغة جافا القياسية، ومجالات تطبيق هذه اللغة قبل استخدام أي من هذه المترجمات.

ولقد اعتمدنا في هذا الكتاب على الأفكار الرئيسية الموجودة في كتاب *Thinking in Java* لمؤلفه الشهير *Bruce Eckel* واستخدمنا الأمثلة الموجودة فيه نظراً لبساطتها وسهولة فهمها. كما استفدنا من العديد من المراجع المذكورة في نهاية الكتاب في

شرح الكثير من الأمور المتعلقة باستخدام جافا في العديد من المجالات كبرمجة قواعد المعطيات وبرمجة أوراكل *Oracle* بشكل خاص، كذلك برمجة الشبكات وغيرها.

سنشرح في **الفصل الثاني** من هذا الكتاب المفاهيم الأساسية للبرمجة غرضية التوجه، حيث سنقوم بشرح الخطوط العامة لهذا النوع من البرمجة، والصفات الأساسية لها كالـ *abstraction* والتوريث *inheritance* وتعديلية الأشكال *polymorphis* وتعديلية النياسب *multithreading* والواجهات *interfaces* وغيرها.

أما في **الفصل الثالث** فسنحاول البدء بالتعرف على هذه اللغة، حيث سنقوم بشرح العناصر الأولية لها، وكيفية بناء الصفوف *classes* وإنشاء الطرق *methods*، وسنقوم بكتابة برنامج بسيط لهذه اللغة وسنشرح أخيراً كيفية توثيق البرامج.

و في **الفصل الرابع** سنوضح تعليمات لغة جافا بشكل مفصل والمعاملات المستخدمة في هذه اللغة مع إعطاء الأمثلة المناسبة.

أما في **الفصل الخامس**، سنقوم بشرح التقنيات التي استخدمتها جافا لحل المشاكل المتعلقة بالقيم الابتدائية والحذف التلقائي لعناصر باستخدام مُجمّع البيانات عديمة النفع *Garbage Collector*.

و في **الفصل السادس**، سنقوم بشرح كيفية استخدام المكتبات والصفوف، وشرح كيفية تركيب الصفوف *Composition* والتوريث منها *Inheritance*.

في **الفصل السابع** سنقوم بإعطاء فكرة مفصلة عن كيفية استخدام خاصية تعددية الأشكال *Polymorphism* في لغة جافا، وكيفية إجراء عملية التوجيه للأعلى *Upcasting*، والصفوف والطرق المجردة *Abstract Classes* و *Methdes*، بالإضافة إلى التوريث المتعدد *Multiple Inheritance*.

في **الفصل الثامن** سنقوم بشرح كيفية استخدام المصفوفات *Arrays*، والأنواع المختلفة للمجموعات *Collection* كالاشعة *Vectors* والعدادات *Enumerations* والمكدسات *Stacks* وغيرها.

في **الفصل التاسع**، سنشرح كيفية قيام جافا بمعالجة الأخطاء باستخدام الاستثناءات *Exceptions* والأنواع الرئيسية للاستثناءات.

في **الفصل العاشر**، سنقوم بشرح مفصل عن نظام الإدخال والإخراج في جافا، وكيفية ضغط البيانات وأرشفة العناصر.

في **الفصل الحادي عشر**، سنقوم بشرح كيفية إنشاء النوافذ والبرمجيات، وعناصر التحكم المعروفة كالأزرار *Buttons* وحقول النص *Text Fields* ومناطق النص *Text Areas* واللصاقات *labels* وصناديق التحقق *Check Boxes* وغيرها. و سنشرح أيضاً كيفية التحكم بتخطيط النوافذ باستخدام الأنماط المعرفية مسبقاً. إضافة إلى كيفية إنشاء نوافذ التطبيقات والقوائم *Menus* وصناديق الحوار *Dialog Boxes*.

أما في **الفصل الثاني عشر** فنسشرح كيفية إجراء البرمجة المرئية *Visual Programming* ضمن لغة جافا وذلك باستخدام حبيبات جافا *Java Beans*.

في **الفصل الثالث عشر** سنقوم بشرح أهم التقنيات الجديدة التي يمكن استخدامها في جافا وهي تقنية تجزيء البرامج إلى مهام فرعية باستخدام تعددية النيااسب *Multithreading*. وسنقوم أيضاً بشرح كيفية القيام بمشاركة المصادر المقيدة، ومجموعات و أفضليات النيااسب.

في **الفصل الرابع عشر**، سنشرح كيفية برمجة الشبكات *network programming* باستخدام لغة جافا. حيث سنشرح أولاً كيفية القيام بتعريف الجهاز، وكيفية التعامل مع المقابس *Sockets*، و تخديم عدة زبائن في نفس الوقت، وسنشرح كيفية استخدام البروتوكول *TCP/IP* ضمن بيئة مختم/زبون *Client/Server*، وأخيراً سنبين فائدة استخدام البروتوكول *UDP*.

في **الفصل الخامس عشر**، سنشرح كيفية القيام بإنشاء تطبيق وب باستخدام لغة جافا، وسنعطي أمثلة بسيطة عن كيفية بناء هذا التطبيق في موقع المختم، وعند الزبون.

أما في **الفصل السادس عشر**، فسنتشرح كيفية استخدام لغة جافا للوصول إلى قواعد المعطيات باستخدام الأداة *JDBC* ومقارنتها مع الأداة *ODBC*، و كيفية برمجة قواعد المعطيات باستخدام هذه اللغة. وسنعطي هنا مثلاً عن كيفية برمجة قواعد معطيات أوراكل *Oracle* بلغة جافا.

أخيراً، و في **الفصل السابع عشر** سنقوم بإعطاء فكرة سريعة عن مكتبة جافا الجديدة *Swing*، حيث سنبين كيفية قلب البرامج القديمة للتمكن من استخدام هذه المكتبة، بالإضافة إلى كيفية إنشاء مختلف عناصر التحكم الأساسية التي رأيناها سابقاً باستخدام إجراءات هذه المكتبة.



نعرف من قبل فإن البرمجة غرضية التوجه *Object Oriented Programming* أو اختصاراً *OOP* تساعدنا في تنظيم البيانات *Data* كما وترتيبها بشكل أقرب إلى الواقع. لذلك فإن العنصر الرئيسي للبرمجة غرضية التوجه هو البيانات، فهي روح أي برنامج. ونفيدنا البرمجة غرضية التوجه في عدة مستويات:

سلسلة الرضا للمعلومات

- ✓ على مستوى المديرين *Managers*: فهي تساعدهم في تطوير وصيانة البرامج بشكل أسرع وأرخص.
- ✓ أما على مستوى المحللين والمصممين *Analysts and Designers*، فيصبح إجراء النمذجة *Modeling Process* أسهل ويعطينا تصميماً أكثر وضوحاً وفعالية.
- ✓ وعلى مستوى المبرمجين *Programmers*، فإن وضوح نموذج العنصر *Object Model* وقوة أدوات البرمجة غرضية التوجه، إضافةً إلى غنى مكتباتها تجعل عملية البرمجة مهمةً مسلية.
- وتختلف طريقة التفكير بالعناصر *Objects* بشكل كبير عن التفكير بالطريقة التقليدية أو الإجرائية *Procedurally*. فلقد كانت الطريقة المتبعة سابقاً في القيام بالبرمجة غرضية التوجه تأخذ أحد منحنيين:
١. اختيار لغة مثل *Smalltalk* تجبرك على تعلم عدد كبير من الدالات قبل أن يصبح بإمكانك إنشاء برنامج متكامل.
 ٢. اختيار لغة أخرى مثل *C++* والتي لا تمتلك افتراضياً أية مكتبة (لحسن الحظ تغير هذا الأمر بشكل واضح الآن بعد إضافة مكتبات الجزء الثالث *Third-Party Libraries* ومكتبة *C++* القياسية *Standard C++ Library*)، وتجبرك على التعمق داخل هذه اللغة حتى تستطيع كتابة مكتبات عناصرك الخاصة.
- لذلك كانت عملية تصميم العناصر صعبة. وكان بعض الخبراء يقومون بتصميم العناصر الأفضل حتى يستطيع الآخرون استخدامها.
- لذلك فإن لغات البرمجة غرضية التوجه الناجحة ليست عبارة عن تركيب نحوي *Syntax* ومترجم *Compiler* فقط، بل هي أيضاً بيئة تطوير كاملة *Development Environment* تتضمن مكتبة مصممة بشكل سهل وبسيط للتعامل مع العناصر. وتصبح المهمة الأساسية لمعظم المبرمجين هي استخدام عناصر موجودة مسبقاً لحل مشاكل تطبيقاتهم.
- سنقوم في هذا الفصل بتوضيح المفاهيم الأساسية للبرمجة غرضية التوجه وذلك باستخدام العديد من أفكار لغة البرمجة *Java* وذلك على المستوى التصميمي *Conceptual*



Level، لكن إياك أن تعتقد بأنه سيصبح بإمكانك كتابة برامج جافا متكاملة عند الانتهاء من قراءة هذا الفصل، وإنما ستحتاج بالطبع لاستكمال قراءة كامل هذا الكتاب!!؟

التجريد Abstraction...

تزدك جميع لغات البرمجة بمستوى ما من التجريد، ويمكن ربط مستوى تعقيد أي مسألة تقوم بحلها بنوعية تجريبها.

فمثلا لغة *Assembly Language* عبارة عن تجريد بسيط للآلة المستخدمة، بينما اللغات التنفيذية *Imperative Language* كلغات *Fortran* و *Pascal* و *C* فهي تجريد للغة *Assembly*. وتعتبر اللغات التنفيذية متطورة إلى حد كبير بالنسبة للغة *Assembly* إلا أن عملية التجريد الأساسية فيها تفرض عليك التفكير ببنية الحاسوب بدلا من التفكير ببنية المسألة التي تقوم بحلها. لذلك يجب على المبرمج تحقيق الارتباط بين نموذج الآلة *Machine Model* ونموذج المسألة *Problem Model* التي سيقوم بحلها. والجهد المطلوب لإنجاز الارتباط السابق سيعطينا برامج صعبة الكتابة وكلفة صيانتها عالية.

سابقا كانت لغات مثل *LISP* و *APL* تختار مشاهد خاصة، حيث أن جميع المسائل إما قوائم *Lists* أو خوارزميات *Algorithmic*. أما لغة *PROLOG* فكانت تقوم بتحويل المسائل إلى سلاسل من القرارات *Decisions Chains*.

أي من اللغات السابقة كانت تقدم حولا جيدة لأنماط خاصة من المسائل، لكن عندما تحتاج إلى الخروج من مجال هذه المسائل فإنها تصبح عاجزة تماما!!!

بينما في النمط غرضي التوجه يمكنك أخذ خطوة أفضل لأنه يقدم للمبرمج أدوات كافية ومناسبة لتمثيل العناصر، لذلك فإن المبرمج لا يصبح مقيدا بأي نمط من أنماط المسائل. ويسمح للبرنامج بتكليف نفسه مع لغة المسألة عن طريق إضافة أنماط جديدة من العناصر. لذلك فعندما تقوم بقراءة الترميز الذي يصف الحل، يمكنك قراءة الكلمات التي تشرح المسألة. هذا يعطيك لغة بتجريد أقوى وأكثر مرونة من اللغات التي ذكرناها سابقا.

يمكن إذا إيضاح الميزات الأساسية لمفهوم البرمجة غرضية التوجه كما يلي:

١. أي شيء عبارة عن عنصر، تذكر بأن العنصر عبارة عن متحول وهمي، يحتوي على بيانات. لكن يمكنك أن تطلب منه إنجاز بعض العمليات على نفسه. نظريا يمكن أخذ أي جزء تصميمي في المسألة التي تقوم بحلها (أبنية، طلاب، خدمات... الخ) وتمثيله كعنصر في برنامجك.

٢. أي برنامج عبارة عن مجموعة من العناصر التي يمكنها أن تخبر بعضها ما الذي يجب أن تفعله وذلك عن طريق الرسائل. فعندما ترغب بإرسال طلب إلى عنصر، تقوم بإرسال رسالة *Send a Message* إلى هذا العنصر. بشكل أوضح، يمكنك اعتبار الرسالة كطلب لتشغيل دالة تنتمي إلى عنصر خاص.

٣. يمتلك أي عنصر ذاكرة خاصة به مكونة من عناصر أخرى، حيث يمكن إنشاء نمط عنصر جديد بإنشاء حزمة تحتوي على عناصر موجودة مسبقا. لذلك يمكنك إخفاء تعقيد برنامج خلف بساطة العناصر.

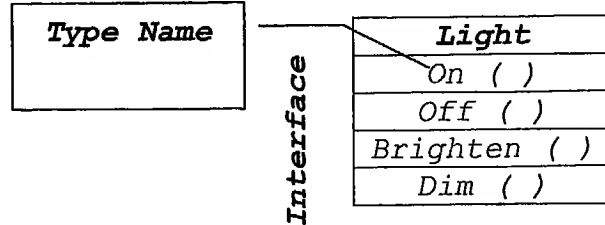
٤. لكل عنصر نمط، حيث أن كل عنصر عبارة عن هيئة تمثل صف *Instance of a Class*. والصف مشابه للنمط، والميزة الأساسية للصف هي ماهية الرسائل التي يمكن إرسالها إليها.

٥. جميع عناصر نمط معين يمكنها استقبال نفس الرسائل، فمثلا أي عنصر من نمط الدائرة *Circle* هو أيضا عنصر من نمط شكل *Shape*، لذلك يجب أن نضمن أن يستقبل عنصر الدائرة نفس رسائل عنصر الشكل. وتعتبر هذه الخاصية أحد أهم ميزات البرمجة غرضية التوجه.

لكل عنصر واجهة إظهار *Interface*...

تعتبر فكرة الربط بين نمط عنصر وبين واجهة إظهار هذا العنصر من القضايا الأساسية في البرمجة غرضية التوجه، فإذا أردنا مثلا التعبير عن مصباح الإضاءة، يمكننا إنشاء

الصف أو النمط *Light*، أما الطلبات التي يمكنك إجراؤها على عنصر *Light* فهي إما إطفائه أو إضاءته أو زيادة أو إنقاص إضاءته.



يمكنك إذا إنشاء مؤشر للنمط *Light* بتعريف اسم لهذا المؤشر (الاسم *It* مثلاً) وإنشاء عنصر من نمط *Light* باستخدام الطريقة *New* على الشكل :

Light It = new Light ()

الآن تستطيع إرسال رسالة إلى هذا العنصر مثلاً: القيام بإطلاقه بتحديد اسم العنصر ثم اسم الرسالة ووضع النقطة كفاصل بينهما :

It.On ()

لذلك فإن الطلبات التي يمكنك إنشاؤها على عنصر يتم تعريفها من خلال واجهة إظهارها، أما النمط فهو ما تحدده هذه الواجهة.

التنفيذ المخفي *The Hidden* ...Implementation

يمكن لجميع منشئات صف *Class Creators* إنشاء أنماط بيانات جديدة. كذلك هنالك مبرمج زبون *Client Programmer* الهدف منه تجميع صندوق أدوات *Toolbox* من الصفوف التي يمكن استخدامها لتطوير التطبيقات بشكل سريع.

أما الهدف من منشئ الصف *Class Creator* فهو بناء صف يقوم بعرض ما هو ضروري فقط لمبرمج الزبون وإخفاء بقية الأجزاء عنه، وبالتالي لا يستطيع استخدام هذه الأجزاء المخفية.

وفي حال إتاحة جميع أعضاء صف *Class Members* لأي كان، يمكن لمبرمج الزبون القيام بأي عمل على هذا الصف، لذلك يفضل في كثير من الأحيان التحكم بالوصول إلى أعضاء صف لسببين:

□ السبب الأول هو عدم إتاحة الفرصة لمبرمج الزبون بالوصول إلى الأجزاء الخاصة بالحسابات الداخلية للآلة.

□ أما السبب الثاني فهو السماح لمصمم المكتبة بتغيير البنية الداخلية دون أن يخشى من كيفية تأثير ذلك على المبرمج الزبون.

تستخدم لغة جافا الكلمات المفاتيح التالية للتحكم بعملية الوصول هذه:

✓ **Public**: تعني بأن التعريف المحدد متاح لأي كان.

✓ **Private**: تعني بأنه لا يمكن لأحد الوصول إلى هذا التعريف ماعداك.

✓ **Protected**: تشبه إلى حد كبير *Private* عدا أن الصف الموروث

inheriting class يمكنه الوصول إلى الأعضاء المحميين
protected members لا الأعضاء المخصصين *private members*.

إعادة استخدام الترميز *Reusing the ...implementation*

بعد أن يتم إنشاء واختبار صف ما، يجب أن يمثل وحدة ترميز مفيدة يمكن إعادة استخدامها فيما بعد. وهذه الخاصية من الخواص المهمة التي تميز البرمجة غرضية التوجه.

الطريقة الأبسط لإعادة استخدام صف هي إعادة استخدام عنصر من هذا الصف مباشرة وهي ما نسميها بإنشاء عنصر عضو *member object*. ويمكن أن يتألف الصف الجديد من أي عدد وأي نمط من العناصر الأخرى. تسمى هذه الخاصية بالتركيب *Composition* وذلك لأنك تقوم بتركيب صف جديد اعتمادا على صفوف موجودة مسبقا. ويمكن ربط هذه الخاصية مع علاقة "تمتلك" أو "has-a".

وتكون عناصر العضو *member objects* في الصف الجديد خاصة *private* حيث لا يستطيع مبرمجي الزبون الوصول إليها. يسمح لك هذا بتغيير الأعضاء دون التأثير على ترميز الزبون الموجود مسبقا. يمكنك أيضا تغيير عناصر العضو أثناء وقت التنفيذ *run time* مما يعطيك مرونة أكثر.

التوريث *Inheritance* ...

نحتاج في كثير من الأحيان إلى إنشاء أنماط وعناصر تتشابه في كثير من الخصائص، إلا أنها قد تختلف في أمور بسيطة، لذلك يمكن استخدام نمط عنصر موجود مسبقا، وإجراء بعض الإضافات أو التعديلات عليه وهو ما نسميه بالتوريث *Inheritance*. نسمي الصف الأساسي بالصف الأب *Parent Class*، أما الصفوف المتفرعة عنه فنسميها بالصفوف الأبناء *Child Classes* أو الصفوف المشتقة *Derived Classes*.

يجب أن نلاحظ بأنه في حال حدوث أي تغيير على الصف الأب فسيؤدي ذلك إلى حدوث تغيير في الصفوف الأبناء. ويتم تنفيذ عملية التوريث في لغة جافا باستخدام كلمة المفتاح *extend*. حيث تقوم بإنشاء صف جديد ثم تقول بأنها توسيع للصف الأساسي. وعندما تقوم بإجراء عملية التوريث فإنك تنشئ نمطا جديدا لا يحتوي على جميع الأعضاء في النمط الجديد فقط وإنما تقوم أيضا بإنشاء نسخة مضاعفة من الصف الأساسي. لذلك فإن جميع الرسائل التي يمكنك إرسالها إلى الصف الأساسي يمكن أيضا إرسالها إلى عناصر

الصفوف الأبناء. كذلك فإن جميع العناصر في الصفوف الأبناء لا تمتلك نفس النمط فقط وإنما لها نفس التصرف *behavior*.

وتوجد طريقة مفيدة تساعدك على التمييز بين الصف الأساسي والصفوف الأبناء، وتعتمد على إضافة دالات جديدة إلى الصفوف الأبناء والتي لن تكون جزءاً من واجهة الصف الأساسي.

هناك طريقة أخرى للتمييز بين الصف الأساسي والصفوف الأبناء، تعتمد على تغيير سلوك دالة الصف الأساسي في الصف الابن وهو ما نسميه بهيمنة الدالة *Function* *Overriding* حيث يتم إنشاء تعريف للدالة في الصف الابن، وكأنك تقوم باستخدام نفس دالة الواجهة لكنك ترغب بصنع شيء مختلف للنمط الجديد.

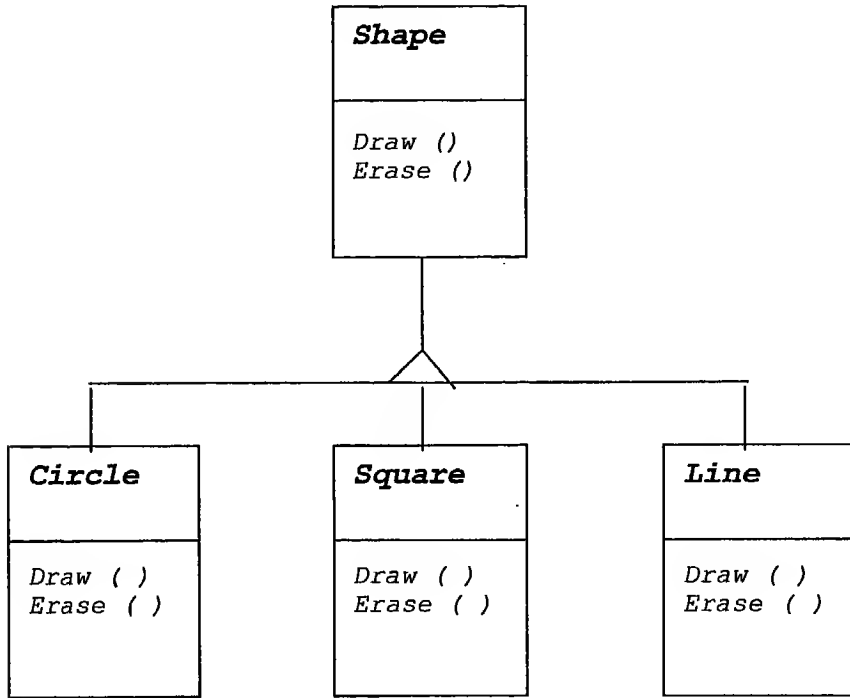
لكن هل يجب على الوراثة أن تهيمن على دالات الصف الأساسي فقط؟ مما يعني بأن النمط المشتق سيكون مشابهاً تماماً لنمط الصف الأساسي لأن له نفس الواجهة. وكنتيجة لذلك يمكن استبدال عنصر من الصف المشتق بعنصر من الصف الأساسي وهو ما يسمى بالاستبدال الصافي *pure substitution* وهي الحالة المثالية للتوريث. أما العلاقة بين الصف الأساسي والصفوف الأبناء فهي من نمط *is-a*، لأنه يمكن القول مثلاً بأن الدائرة هي شكل *"a circle is a shape"*.

وفي بعض الحالات تحتاج إلى إضافة عناصر إظهار جديدة للنمط المشتق وذلك بتوسيع الواجهة وإنشاء نمط جديد. ويمكن أن يتم استبدال النمط الجديد بالنمط الأساسي، لكن عملية الاستبدال هذه ليست مثالية ويمكن وصف هذه العلاقة من نمط *is-like*. ففي هذه الحالة يمكن للنمط الجديد أن يأخذ نفس واجهة النمط القديم لكنه يمتلك أيضاً دالات أخرى، لذلك لا يمكنك القول بأنه مشابه تماماً للنمط الأصلي.



تعددية الأشكال *Polymorphism*...

كما لاحظنا فإن الوراثة تساعد في إنشاء عائلة صفوف تعتمد كلها على نفس الواجهة النظامية. لنأخذ المخطط التالي :



وأحد الأمور الهامة التي يمكن القيام بها مع هذا النوع من عائلة الصفوف هو معالجة الصف المشتق كعنصر من الصف الأساسي. هذا الأمر هام لأنه يعني أنك تستطيع كتابة جزء ترميز يتجاهل التفاصيل الخاصة بالنمط ويمكنه مخاطبة الصف الأساسي. فإذا تمّت إضافة نمط جديد *Triangle* مثلاً وذلك باستخدام التوريث، فإن الترميز الذي ستركبه

سيعمل ضمن هذا النمط الجديد كما في بقية الأنماط الأخرى الموجودة وبشكل مناسب. لذلك فإن هذا سيكون البرنامج قابلاً للتوسيع *extensible*.
لنأخذ المثال السابق ولنطبق عليه الدالة التالية (المكتوبة بلغة جافا):

```
void doStuff (Shape s) {
    s.erase ( );
    // ...
    s.draw ( );
}
```

يمكن للدالة السابقة أن تتخاطب مع أي شكل من نمط *Shape*، لذلك فهي غير مرتبطة بنمط العنصر الذي ستقوم برسمه أو بحذفه (سواءً أكان هذا العنصر دائرة أو مربع أو مستقيم). أما إذا استخدمنا هذه الدالة كما في المثال التالي:

```
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
```

ستلاحظ بأن استدعاء الدالة *doStuff ()* سيعمل بشكل صحيح تلقائياً بغض النظر عن النمط الفعلي للعنصر.
لنأخذ مثلاً السطر:

```
doStuff(c);
```

نلاحظ هنا بأن مؤشراً من نمط دائرة *Circle* قد تم تمريره إلى دالة تنتظر مؤشراً من نمط شكل *Shape*. وعلى اعتبار أن الدائرة عبارة عن شكل *Circle* is a *Shape* "يمكن إذاً معالجتها باستخدام هذه الدالة. لذلك فإن أية رسالة يمكن للدالة إرسالها إلى عنصر من نمط *Shape* سيتمكن العنصر من النمط *Circle* قبولها أيضاً، وهو أمر منطقي تماماً.

تدعى عملية معالجة نمط مشتق كما لو أنه في النمط الأساسي بالتوجيه للأعلى *Upcasting* وذلك للدلالة على توجيه التوريث نحو الأعلى.
سأريك شيئاً مسلياً أكثر...



عند طلب الدالة () Draw على عنصر من نمط دائرة Circle، فإن ذلك سيتطلب تنفيذ ترميز معين يختلف عن ذلك المتعلق بعنصر من نمط مربع Square أو مستقيم Line.

لكن عند إرسال () Draw إلى عنصر من نمط شكل Shape مجهول الهوية، فإن السلوك الصحيح لهذه الرسالة لن يتحدد إلا بتحديد النمط الحالي المطبق.

وهذا شيء رائع بالفعل، لأن مترجم جافا Java Compiler وعند قيامه بترجمة ترميز الدالة () doStuff فإنه لا يعرف تماماً نمط العنصر الذي يتعامل معه. لذلك سيحاول استدعاء نسخة () erase ونسخة () draw الموجودتان ضمن الصف Shape وليست تلك الموجودة في الأنماط الخاصة.

تدعى عملية إرسال رسالة إلى عنصر محدد دون أن نعرف نمطه بتعددية الأشكال Polymorphism، أما الإجراء المستخدم من قبل لغات البرمجة غرضية التوجه لتنفيذ تعددية الأشكال فنسميه بالربط الديناميكي Dynamic Binding.

تحتاج بعض اللغات إلى استخدام كلمة مفتاح لتفعيل الربط الديناميكي، فتستخدم لغة ++C مثلاً كلمة المفتاح Virtual. أما في لغة جافا فليست بحاجة إلى استخدام أي كلمة مفتاح وتتم هذه العملية بشكل تلقائي.

الصفوف الأساسية المجردة Abstract

...Base Classes

في كثير من الأحيان، وعند قيامك بتصميم تطبيق، تحتاج فقط إلى إظهار الصفوف المشتقة للصف الأساسي. أي أنك لا ترغب بالقيام بإنشاء عنصر في الصف الأساسي وإنما إجراء عملية التوجيه إلى الأعلى upcast بحيث يمكنك استخدام واجهته. تستطيع القيام بذلك بجعل هذا الصف مجرداً وذلك باستخدام كلمة المفتاح abstract، فإذا حاول أيّاً كان توليد عنصر في الصف المجرد فإن المترجم سيمنعه من القيام بذلك.

يمكنك أيضاً استخدام كلمة المفتاح *abstract* لتوصيف طريقة *method* لم يتم تنفيذها بعد، أي لم يتم تحديد عملها. ولا يمكن إنشاء طريقة مجردة *abstract method* إلا ضمن صف مجرد *abstract class*. لكن عندما يتم توريث الصف، فإنه يجب تنفيذ أو توصيف عمل الطريقة، وإلا فإن الصف المشتق سيصبح مجرداً كذلك.

وهناك كلمة مفتاح أخرى *interface* تأخذ بمفهوم الصف المجرد خطوة أبعد وذلك بمنع تعريف أي دالة بشكل كامل. لذلك فهي مفيدة من أجل الفصل بين الواجهة *interface* والتنفيذ *implementation*، كما أن باستطاعتك الربط بين عدة واجهات.

مناظر وأعمار العناصر *Object Landscapes and Lifetimes*

من الأمور الهامة جداً في البرمجة غرضية التوجه هي كيفية إنشاء وإلغاء العناصر، كذلك مكان موضع بيانات العنصر وكيفية التحكم بعمر هذا العنصر.

توجد العديد من الفلسفات التي يمكن أخذها بعين الاعتبار، فمثلاً تقوم لغة *C++* بإعطاء الخيار للمبرمج من أجل القيام بذلك. لذلك يمكن تحديد مكان تخزين بيانات العنصر وعمره عند كتابة البرنامج، وذلك بوضع هذا العنصر على مكثس *Stack* أو في منطقة تخزين ثابتة *static storage area*. طبعاً أنت هنا تضحي بمرونة البرنامج لأنه يتوجب عليك معرفة العدد الفعلي للعناصر وأعمارها وأنماطها أثناء قيامك بكتابة البرنامج.

أما الفلسفة الأخرى فتقوم على توليد العناصر تلقائياً في حوض ذاكرة يسمى الكومة *Heap*. لا تستطيع هنا معرفة عدد العناصر التي تحتاجها، وأنماط هذه العناصر وأعمارها إلا أثناء وقت التنفيذ *run time*.

تسمح لك لغة *C++* بتحديد إنشاء العناصر عندما تقوم بكتابة البرنامج أو أثناء وقت التنفيذ.

أما لإلغاء عنصر فهناك خياران، الأول هو تحديد متى يتم إلغاء العنصر برمجياً، أو متى يمكن استخدام ما يسمى بمجمّع النفايات *Garbage Collector* الذي يقوم تلقائياً باستكشاف العناصر التي لم تعد تستخدم ويقوم بحذفها. مجمّع النفايات *Garbage Collector* هذا غير موجود في لغة *C++* لكن يمكنك استخدامه في لغة جافا. أليس هذا الأمر مفيداً ؟!!

المجموعات والتكرارات *Collections* ...and *Iterators*

إذا لم تعرف عدد العناصر التي ستحتاجها في حل مسألة معينة، أو مدّة استخدام هذه العناصر، أو كيفية تخزينها. فإن الحل الذي يمكنك استخدامه في البرمجة غرضية التوجه يتمثل بإنشاء نمط عنصر جديد يساعدك على حل هذه المسألة ويمكنه الربط مع بقية العناصر، يسمى هذا العنصر الجديد بالمجموعة *Collection*، وهو يقوم بتوسيع نفسه عند الضرورة لذلك فلن نعد بحاجة إلى معرفة عدد العناصر التي ستتضمنها المجموعة.

يأتي هذا النمط من العناصر مع لغات البرمجة غرضية التوجه القوية. فهو في لغة *C++* مثلاً عبارة عن *Standard Template Library (STL)*، أما في لغة جافا فهي متضمنة في مكتبتها القياسية، وفي لغة باسكال الغرضية *Object Pascal* فهي موجودة في *Visual Component Library (VCL)*. ويمكن للمجموعات أن تأخذ أنماطاً مختلفة فقد تكون عبارة عن أشعة *Vectors* أو لوائح مرتبطة *Linked Listes* أو أشجار *Trees* أو مكّنسات *Stacks*...الخ.

وحتى تستطيع معالجة عدة عناصر ضمن مجموعة، يجب عليك استخدام ما يسمى بال تكرارات *Iterators*، وهي عبارة عن عناصر تقوم باختيار عناصر أخرى من مجموعة.

الشيء الهام الذي يتوجب عليك معرفته هو أن التكرار يسمح لك بعبور المجموعة دون النظر إلى بنية هذه المجموعة سواءً أكانت شعاعاً أو لائحة مرتبطة أو شجرة... الخ. هذا الأمر يعطيك مرونةً عن طريق إمكانية تغيير بنية المعطيات دون تخريب ترميز برنامجك. تحتوي لغة جافا في نسخها القديمة (1.0 و 1.1) على تكرار بالاسم *Enumeration*، أما في النسخ الجديدة (1.2 وأحدث) فقد تمت إضافة التكرارات *Iterator* إلى مكتبتها.

يوجد سببان لاختيار المجموعات:

✓ السبب الأول هو أنها تزدك بأنماط عديدة من الواجهات *Interfaces* والسلوك الخارجي *External Behaviors*، فالمكدس *Stack* له واجهة وسلوك مختلف عن الكومة *Heap* والتي تمتلك بدورها واجهة وسلوكاً مختلفاً عن اللائحة *List* وهكذا...

✓ أما السبب الثاني فهو أن المجموعات المختلفة لها أداء مختلف لعمليات محددة. فمثلاً الشعاع *Vector* واللائحة *List* هما سلاسل بسيطة ولهما نفس الواجهة ونفس السلوك الخارجي، لكن بعض العمليات قد تكون مختلفة من ناحية الأداء، فالوصول العشوائي *Random Access* إلى العناصر في شعاع يأخذ زمناً ثابتاً بغض النظر عن العنصر الذي تختاره. بينما في اللوائح المرتبطة فإنها تأخذ وقتاً طويلاً لاختيار العناصر عشوائياً.

لذلك وفي مرحلة التصميم *Design Phase* يمكنك البدء بالتعامل مع لائحة *List*، وعند الحاجة إلى تحسين الأداء قم بالتغيير إلى شعاع *Vector*.



معالجة الاستثناءات *Exception Handling*...

منذ بداية لغات البرمجة، اعتبرت مشكلة معالجة الأخطاء من أهم وأصعب المشاكل التي توجب حلها. والسبب في ذلك صعوبة تصميم برنامج خالٍ تماماً من الأخطاء. وتتجاهل الكثير من لغات البرمجة هذه المشكلة، حيث تترك للمبرمج البحث عن حل لها ولوحده. بينما تقوم بعض اللغات باستخدام ما يسمى بمعالجات الأخطاء *Exception Handling* والتي تحاول اكتشاف الأخطاء في لغات البرمجة وحتى في نظام التشغيل أحياناً.

وتكون الاستثناءات عبارة عن عناصر *Objects* يتم بثها من موقع الخطأ ويمكن بعدها لمعالج الاستثناء التقاطها في حال حدوث نمط معين من الأخطاء. وفي لغة جافا تعتبر عملية معالجة الاستثناءات إجبارية. وإذا لم تقم بكتابة البرامج بطريقة تضمن معالجة الاستثناءات بشكل سليم فستحصل على رسائل أخطاء أثناء عملية الترجمة.

تعدد النيات *Multithreading*...

من الأمور الهامة المرتبطة بلغات البرمجة هي فكرة معالجة أكثر من مهمة في نفس الوقت. والعديد من مسائل البرمجة تتطلب أن يتمكن البرنامج من إيقاف ما يقوم بعمله ومن ثم معالجة مسألة أخرى ثم العودة إلى المسألة الأساسية. وهناك العديد من الحلول لهذه المشكلة، منها ما كان يستخدم في العديد من لغات البرمجة التقليدية، والتي تحاول استخدام ما يسمى بخدمة المقاطعة *Interrupt Service*، لكنها طريقة صعبة تتطلب خبرة ومعرفة في كيفية استخدام مقاطعات الآلة *Machine Interrupts*.

من الحلول الأخرى أيضاً القيام بتقسيم المسألة إلى أجزاء تنفيذية منفصلة تسمى النياسب من *Threads*. من الأمثلة العملية على ذلك واجهة المستخدم *User Interface* حيث يمكن للمستخدم عن طريق النياسب الضغط على زر والحصول على جواب سريع بدلاً من إجباره على الانتظار حتى يقوم البرنامج بإنهاء مهمته الحالية. ولقد تمّ بناء النياسب ضمن لغة جافا بحيث يمكنها تسهيل معالجة الكثير من المسائل المعقّدة. وتكون على مستوى العنصر عادةً، لذلك يمكن تمثيل نيسب تنفيذي بعنصر واحد. كما تزود هذه اللغة بإمكانية قفل محدّدات المصادر *Limited Resource* *Locking*، بحيث تستطيع قفل ذاكرة أي عنصر ليتمكن نيسب وحيد من استخدامه في وقت معيّن، ويتم ذلك باستخدام كلمة المفتاح *Synchronized*. أو باستخدام أنماط أخرى من المصادر من قبل المبرمج بشكل صريح، حيث يقوم بإنشاء عنصر يمثل القفل *Lock* وتقوم جميع النياسب بالتحقق منه قبل الوصول إلى هذه المصادر.





على الرغم من أن لغة جافا تعتمد بشكل رئيسي على لغة ++C، إلا أنّها لغة غرضيّة التوجّه بشكل فعلي. فلغة ++C كما نعرف عبارة عن توسعة للغة الأساسية C، وهذا ما أدى إلى وجود بعض الأمور غير المقبولة في هذه اللغة وسبّب تعقيدها.

بينما نفترض لغة جافا بأنك ترغب فقط بالبرمجة غرضية التوجه. هذا يساعدك على إمكانية البرمجة بلغة بسيطة التعلم والاستخدام نسبة للغات البرمجة غرضية التوجه الأخرى.

سنقوم في هذا الفصل بالتعرف على المكونات الأساسية لبرنامج جافا، وستعرف بأن أي شيء في جافا، حتى البرنامج نفسه، عبارة عن عنصر.

يمكنك التعامل مع العناصر باستخدام المؤشرات...

لكل لغة برمجة مفهومها الخاص في معالجة المعطيات. فقد يخشى المبرمج بشكل دائم من نمط المعالجة المستخدم، أو أنه يقوم بمعالجة العنصر مباشرة أو التعامل مع ممثل غير مباشر لهذا العنصر (كالمؤشرات في لغة C أو C++) يحتاج إلى تركيب نحوي Syntax خاص.

كل ما ذكرناه سابقاً يمكن إجراؤه بشكل بسيط في جافا. فأنت تتعامل مع أي شيء على أنه عنصر *Object*، لذلك هناك تركيب متماسك وحيد تستخدمه في أي مكان. لكن على الرغم من أنك تتعامل مع أي شيء على أنه عنصر، إلا أن المحدد *Identifier* الذي تعالجه عبارة عن مؤشر *Handle* نحو هذا العنصر. يمكنك أن تتخيل هذا الأمر على النحو التالي:

لنفترض أن العنصر لديك هو جهاز التلفزيون، وأن المؤشر عبارة عن جهاز التحكم *Remote Control*. فطالما أنك تحمل جهاز التحكم فبإمكانك وبسهولة إجراء الاتصال مع جهاز التلفزيون، وعندما يطلب منك أحد ما تغيير القناة أو رفع الصوت فأنت هنا تتعامل مع المؤشر (جهاز التحكم) الذي يقوم بدوره بتعديل العنصر (التلفزيون). وعندما ترغب بالانتقال ضمن الغرفة والتحكم بالتلفزيون، يجب أن تحمل معك جهاز التحكم (المؤشر) وليس جهاز التلفزيون.

يمكنك كذلك امتلاك جهاز تحكم بدون أن يكون لديك جهاز تلفزيون. لذلك عندما يكون لديك مؤشر *Handle* فهذا لا يعني بالضرورة أن يكون هناك عنصر متصل به. من أجل ذلك عندما تريد احتواء كلمة أو جملة يجب عليك إنشاء مؤشر من نمط *String*:
`String s;`
 لكن هنا قمنا بإنشاء المؤشر لا العنصر. وعندما تقرر أن ترسل رسالة إلى *s* عند هذه النقطة ستحصل على رسالة خطأ (أثناء وقت التنفيذ) لأن *s* غير متصل حالياً بأي عنصر. الحل الأفضل دوماً هو وضع قيمة ابتدائية للمؤشر عند إنشائه، مثلاً:
`String s="mouhib";`

يجب عليك إنشاء جميع العناصر...

عندما تقوم بإنشاء مؤشر، يتوجب عليك ربطه بعنصر جديد. تستطيع القيام بذلك باستخدام كلمة المفتاح *new*، لذلك يمكنك باستخدام المثال السابق القول:
`String s = new String ("mouhib");`
 في هذه الحالة لن يتم إنشاء عنصر جديد فقط، وإنما وضع قيمة ابتدائية في هذا العنصر. بالطبع، فإن *String* ليس النمط الوحيد الموجود، وإنما هنالك أنماط أخرى يمكنك استخدامها. الأمر الأهم هنا هو كيف سيكون بإمكانك إنشاء أنماط الخاصة، لأن هذه العملية تعتبر من الأمور الأساسية عند البرمجة بلغة جافا. وهو ما سنحاول تعليمك إياه في الفصول القادمة.

لكن أين يتم تخزين البيانات؟

في لغة جافا، تتوفر لديك ستة أماكن لتخزين البيانات هي:
 ١. المسجلات *Registers*: وهي التخزين الأسرع لأنها تتواجد في مكان مختلف عن بقية أماكن التخزين الأخرى وذلك داخل المعالج *processor*. لكن عدد المسجلات التي يمكنك حجزها مقيد جداً، لذلك يتم حجزها من قبل المترجم

Compiler وفقاً لحاجته. ولا توجد لديك أية إمكانية للتحكم بالمسجلات ولا حتى معرفة وجودها في برامجك أصلاً.

٢. المكدس *Stack* : ويتواجد عادةً في ذاكرة الولوج العشوائي *RAM* لكي المعالج يستطيع التعامل معه مباشرةً من خلال مؤشر المكدس *Stack Pointer*. وينتقل المؤشر نحو الأسفل لإنشاء ذاكرة جديدة، كما ينتقل نحو الأعلى لتحرير هذه الذاكرة. ويجب إعلام مترجم جافا *Java Compiler*، عند إنشاء البرنامج، عن الحجم الصحيح لجميع البيانات التي سيتم تخزينها في المكدس وعمر هذه البيانات، وذلك لأنه يتوجب عليه توليد الترميز المناسب لتحريك مؤشر المكدس نحو الأعلى ونحو الأسفل.

٣. الكومة *Heap*: وهي عبارة عن وعاء *Pool* موجود في الذاكرة (أيضاً في ذاكرة *RAM*) حيث تعيش جميع عناصر جافا. الشيء الجميل هنا، وبالعكس المكتسبات، أن المترجم لا يحتاج إلى معرفة الحجم التخزيني المطلوب حجزه في الكومة ولا حتى عمر البيانات التي سيتم وضعها فيها. لذلك توجد مرونة كبيرة في التعامل مع هذا النوع من أنواع تخزين البيانات. ودائماً عندما تكون بحاجة إلى إنشاء عنصر جديد، يمكنك ببساطة كتابة ترميز إنشاء هذا العنصر باستخدام *New*، يتم بعدها حجز مكان لهذا العنصر على الكومة عند تنفيذ هذا الترميز، علماً أنك ستحتاج إلى زمن أكبر عند حجز هذه الكومة.

٤. التخزين الساكن *Static Storage*: ونعني هنا بالسكون *Static* المكان الثابت *fixed location* (وهو موجود في ذاكرة *RAM* أيضاً). ويحتوي التخزين الساكن على البيانات التي ستبقى متاحة طوال فترة تنفيذ البرنامج.

٥. التخزين الثابت *Constant Storage*: يتم غالباً وضع القيم الثابتة ضمن ترميز البرنامج مباشرةً والذي يعتبر عامل أمان لأنه لا يمكن تغييرها أبداً. ويمكن وضعها أحياناً وبشكل اختياري في ذاكرة القراءة فقط *ROM*.

٦. التخزين خارج ذاكرة الولوج العشوائي *Non-Ram Storage*: عندما تكون البيانات خارج البرنامج بشكل كامل، يمكن عندها أن تتواجد عندما لا يكون البرنامج في حالة عمل، وخارج سيطرة هذا البرنامج. المثالان الأساسيان هنا:



- العناصر المتقاطرة *Streamed Objects*: حيث تتحول العناصر إلى أمواج من البايتات يتم إرسالها عادةً إلى جهاز آخر.
- العناصر الدائمة *Persistent Objects*: حيث يتم وضع العناصر على قرص التخزين *Disk* بحيث يمكن التعامل معها حتى لو انتهى البرنامج.

الأمر الهام هنا هو أن هذا النمط من التخزين يقوم بتحويل العناصر إلى شيء يمكنه أن يتواجد على وسائط أخرى، كما يمكن أن ترسل إلى عناصر ضمن ذاكرة الولوج العشوائي *RAM* عند الحاجة.

الأنماط الأولية...

هنالك مجموعة من الأنماط التي تتعامل معها غالباً ضمن برامجك، وعليك معاملتها معاملة خاصة، والسبب في ذلك هو أنك عندما تقوم بإنشاء عنصر صغير (متحول بسيط) باستخدام *new*، فلن يكون الأمر فعالاً لأن عملية *new* تقوم بوضع عنصر جديد فوق الكومة *Heap*.

لذلك ومن أجل الأنماط البسيطة فقط، فإن جافا تأخذ نفس التقنية المستخدمة في لغة *C* أو *C++* والتي تقوم على إنشاء متحول بشكل تلقائي دون الحاجة إلى استخدام *new*، لذلك فهو لن يكون مؤشراً لأي عنصر، ويمكنه أن يحتوي على قيمة ويتم وضعه على المكّس مما يجعله أكثر فعالية.

وتحدّد جافا حجم كل نمط أولي *Primitive Type*. ولن يتغيّر هذا الحجم من آلة لأخرى كما في أغلب اللغات، وهذا هو أحد الأسباب التي تجعل من لغة جافا لغة محمولة *portable*.

يوضح الجدول التالي الأنماط الأولية في لغة جافا، وحجم كل نمط، كذلك القيمة الصغرى والقيمة العظمى التي يمكن أن يأخذها كل نمط، إضافةً إلى النمط المغلف *Wrapper* *Type* لكل نمط أولي.

النمط الأولي	الحجم	القيمة الصغرى	القيمة العظمى	النمط المغلف
<i>boolean</i>	1-bit	-	-	<i>Boolean</i>
<i>char</i>	16-bit	Unicode 0	Unicode 216-1	<i>Character</i>
<i>byte</i>	8-bit	-128	+127	<i>Byte</i>
<i>short</i>	16-bit	-2^{15}	$+2^{15}-1$	<i>Short</i>
<i>int</i>	32-bit	-2^{31}	$+2^{31}-1$	<i>Integer</i>
<i>long</i>	64-bit	-263	$+2^{63}-1$	<i>Long</i>
<i>float</i>	32-bit	IEEE754	IEEE754	<i>Float</i>
<i>double</i>	64-bit	IEEE754	IEEE754	<i>Double</i>
<i>void</i>	-	-	-	<i>Void</i>

ملاحظة: يفيدك النمط المغلف عندما تحتاج إلى جعل عنصر غير أولي موجود أعلى الكومة يمثل عنصراً أولياً، في هذه الحالة تستخدم النمط المغلف الموافق، مثلاً:

```
char c='x';
```

```
Character C = new Character( c );
```

أو بإمكانك استخدام:

```
Character C = new Character ('x');
```

ملاحظة: أضافت لغة جافا، اعتباراً من الإصدار 1.1، صنفين جديدين لمعالجة الأرقام ذات الدقة العالية جداً هما:

١. *BigInteger*: يدعم الأرقام الصحيحة ذات الدقة الاعتباطية - *arbitrary*

precision intergers. أي أن باستطاعتك وبشكل دقيق تمثيل القيم

الصحيحة بأي حجم دون إضاعة أيّة معلومات أثناء إجراء العمليات عليها.

٢. *BigDecimal*: ويدعم الأرقام ذات الدقة الاعتباطية وبنقطة ثابتة

arbitrary-precision fixed-point numbers. وتستطيع

استخدام هذا النمط لحسابات العملات مثلاً.

المصفوفات في جافا...

كما نعرف فإن أغلب لغات البرمجة تدعم المصفوفات *Arrays*. واستخدام المصفوفات في لغة *C* أو لغة *C++* محفوف بالمخاطر، والسبب في ذلك هو أن هذه المصفوفات عبارة عن كتل من الذاكرة فقط. لذلك في حال قيام البرنامج بمحاولة كتابة في مصفوفة خارج نطاق كتل ذاكرتها أو عند استخدام الذاكرة قبل تهيئةها *Initialization* (والتي هي عبارة عن أخطاء شائعة) فستحصل بالتأكيد على نتائج غير متوقعة.

وكما ذكرنا ونكرر دائماً بأن الغاية الأساسية من بناء لغة جافا هو الأمان *Safety*، لذلك فإن أغلب المشاكل التي يبتلي بها المبرمجون في *C* أو *C++* لن تواجههم في جافا. حيث تضمن لك جافا تهيئةها ولن يتم الوصول إلى خارج نطاق مجالها.

عندما تقوم بإنشاء مصفوفة عناصر، فأنت تقوم بإنشاء مصفوفة مؤشرات فعلياً، وتتم تهيئة كل من هذه المؤشرات تلقائياً بقيمة خاصة هي *null*. وعندما يرى البرنامج القيمة *null* فهو يعرف بأن المؤشر لا يدل على أي عنصر. لذلك يجب ربط عنصر بكل مؤشر قبل استخدامه. وعندما تحاول استخدام مؤشر *null* فإن المشكلة ستظهر أثناء وقت التنفيذ. لذلك فإن الأخطاء الاعتيادية غير مسموحة في جافا. في الفصول القادمة سترى شرحاً مفصلاً عن المصفوفات.

لن تكون بحاجة أبداً إلى تدمير عنصر...

في أغلب لغات البرمجة، تأخذ قضية عمر *lifetime* متحول حيزاً هاماً من العمل البرمجي، حيث نحتاج إلى الإجابة عن العديد من التساؤلات، كالمدة التي يحتاجها هذا المتحول، أو متى يتوجب علينا تدميره، وغير ذلك من التساؤلات التي يمكن أن تسبب العديد من العلال *bugs*.

سنوضح لك في الفقرات القادمة كيف تساعدك لغة جافا على تبسيط المشاكل السابقة وذلك بإجراء جميع عمليات التنظيف والتدمير الضرورية.

لنتعرف أولاً على مفهوم نطاق العمل...

تمتلك أغلب لغات البرمجة الإجرائية *Procedural Languages* ما يسمى بمفهوم نطاق العمل *scope* والذي يحدّد عمر *lifetime* ورؤية *visibility* الأسماء.

يتحدّد نطاق العمل في لغات مثل *C* و *C++* و *Java* باستخدام الأقواس الكبيرة { } مثلاً:

```
{
    int x = 12;
    /* only x available */
    {
        int q = 96;
        /* both x & q available */
    }
    /* only x available */
    /* q "out of scope" */
}
```

ويجب ملاحظة أن المتحول الذي يتم تعريفه ضمن نطاق عمل سيكون متاحاً فقط حتى نهاية هذا النطاق.

ملاحظة: المثال التالي غير مقبول في لغة جافا على الرغم من أنه مقبول في *C* و *C++*:

```
{
    int x = 12;
    {
        int x = 96; /* illegal */
    }
}
```

وسيقبلك المترجم *Compiler* بأن المتحول *x* قد تمّ تعريفه من قبل. لذلك فإن قابليّة *C* و *C++* على إخفاء المتحولات غير مسموحة في لغة جافا لأنّ هذا سيؤدي إلى حدوث تشويش ضمن البرنامج.



لنتعرف الآن على نطاق عمل العناصر...

لا تمتلك عناصر جافا نفس عمر المتحولات الأولية. وعندما تقوم بإنشاء عنصر جافا باستخدام *new* كما في المثال:

```
{
    String s = new String("a string");
} /* end of scope */
```

فإنّ المؤشر *s* سيموت في نهاية نطاق العمل. أما العنصر *String* الذي يدلّ عليه المؤشر *s* فإنّه يبقى مستقراً للذاكرة.

لذلك فإنّه لا توجد في الترميز السابق أية طريقة للوصول إلى هذا العنصر، لأنّ المؤشر الوحيد الدالّ عليه موجود في نهاية النطاق.

لكن السؤال الهام الذي يطرح هنا هو: بما أنّ جافا تترك العناصر تسرح وتمرح في الذاكرة، ألا يؤدي هذا إلى توقّف البرنامج.

الجواب هنا هو أنّ هذا الأمر قد يحدث في *C++* لكن في *Java* فالجواب لا بالتأكيد. والسبب هو أنّ جافا تمتلك ما يسمى بمجمّع النفايات *Garbage Collector* والذي يقوم بالبحث عن العناصر التي تمّ إنشاؤها باستخدام *create* ولا يوجد أيّ مؤشر يدلّ عليها. عندها يقوم بتحرير الذاكرة التي حجزها لهذا العنصر وجعلها متاحة لعناصر جديدة.

وببساطة لا تقلق أبداً فهناك من يقوم بتحرير الذاكرة عنك.

هذا الأمر يلغي بعض مشاكل البرمجة خاصة تلك المتعلقة بضياع الذاكرة *memory leak* والناجمة عن نسيان المبرمج تحرير الذاكرة لديه في كثير من الأحيان.

إنشاء الصفوف...

إذا كان أيّ شيء عبارة عن عنصر، فما الذي يحدّد كيف سيبدو صفّاً خاصاً من العناصر، وكيف سيتصرّف هذا الصف. بمعنى آخر ما الذي يحدّد نمط هذا العنصر.

في أغلب لغات البرمجة غرضية التوجه يتم استخدام كلمة المفتاح `class` لتحديد نمط العنصر الجديد، لنأخذ المثال التالي:

```
class ATypeName { /* class body goes here */ }
```

سيؤدي هذا إلى إنشاء نمط جديد، وبإمكانك الآن إنشاء عنصر جديد من هذا النمط على الشكل:

```
ATypeName a = new ATypeName();
```

طبعاً يتكوّن جسم هذا الصف من تعليق لهذا فهو لن يفيدك كثيراً.

بناء الحقول...

عندما تقوم بتعريف صف ، فإنه باستطاعتك وضع نمطين من العناصر في هذا الصف:

١. المعطيات الأعضاء `Data Members` والتي تسمى الحقول `Fields`.

٢. الدالات الأعضاء `Member Functions` وتسمى عادةً الطرق `Methods`.

أما المعطيات الأعضاء `Data Members` فهي عبارة عن عناصر من أي نمط (يمكنك مخاطبتها من خلال مؤشراتهما) أو قد تكون أحد الأنماط الأولية `Primitive Types` (والتي ليس لها مؤشرات).

فإذا كانت مؤشرات لعناصر، يجب أولاً وضع قيمة ابتدائية لها لربطها بالعنصر الحالي باستخدام دالات خاصة تسمى البانيات `Constructors`.

أما إذا كانت من نمط أولي `primitive type` فيمكنك وضع قيمة ابتدائية لها وذلك مباشرة عند تعريفها في الصف.

ويحتفظ كل عنصر بمنطقة تخزين خاصة به من أجل المعطيات الأعضاء التي يمتلكها، وهي غير مشاركة بين العناصر.

لنأخذ الصف التالي كمثال:

```
class DataOnly {
    int i;
    float f;
    boolean b;
```



}

هذا الصف لا يقوم بأي شيء، لكن باستطاعتك إنشاء عنصر على الشكل:

```
DataOnly d = new DataOnly();
```

تستطيع الآن تحديد قيم للمعطيات الأعضاء في الصف، لكن عليك أولاً معرفة كيفية الوصول إلى أعضاء عنصر. يمكنك القيام بذلك من خلال تحديد اسم مؤشر العنصر متبوعاً بنقطة متبوعاً باسم العضو المحدد ضمن العنصر، مثلاً:

```
d.i = 47;
```

```
d.f = 1.1f;
```

```
d.b = false;
```

ملاحظة: عند استخدام نمط معطيات أولي *primitive data type* كعضو في صف، فإنه يأخذ قيمة افتراضية موضحة في الجدول التالي:

النمط الأولي	القيمة الافتراضية
Boolean	False
Char	'\u0000' (null)
Byte	(byte) 0
Short	(short) 0
Int	0
Long	0L
Float	0.0f
Double	0.0d

هذه القيم الافتراضية تأخذ فقط عند اختيار النمط العضو في صف وليس متحولاً محلياً.

فمثلاً عند تعريف متحول ضمن دالة كما في الشكل:

```
int x;
```

فإن x سيأخذ قيمة عشوائية (كما في C و $C++$) ولن يأخذ القيمة الابتدائية 0.

طبعاً ستكون أنت المسؤول عن وضع قيمة مناسبة قبل استخدام x . فإذا نسيت فإن لغة جافا (وهذا أيضاً تحسین على لغة $C++$) ستعطيك رسالة خطأ أثناء وقت التنفيذ تخبرك بأنه لم يتم وضع قيمة ابتدائية للمتحول x .

الطرق Method...

حتى الآن، وفي لغات البرمجة الإجرائية خاصة، استخدمنا تعبير الدالة *Function* للدلالة على اسم إجراء. أما في لغات البرمجة غرضية التوجه، وفي جافا بشكل خاص، فإن التعبير المستخدم هو الطريقة *Method*. والتعبيران المستخدمان يدلان في الواقع على نفس الشيء.

في لغة جافا تحدد الطريقة *Method* الرسالة التي يمكن أن يتلقاها أي عنصر. وتأخذ عادة الشكل:

```
returnType methodName( /* argument list */ ) {
    /* Method body */
}
```

يمكن إنشاء الطرق في جافا كجزء من الصف فقط، كما يمكن طلب الطريقة من أجل عنصر محدد فقط، ويجب أن يكون هذا العنصر قادراً على تنفيذ هذه الطريقة. يمكنك استدعاء طريقة على عنصر بكتابة اسم العنصر متبوعاً بنقطة متبوعاً باسم الطريقة وقائمة وسطائها كما في الشكل:

```
ObjectName.methodName(arg1, arg2, arg3)
```

فإذا كان لدينا مثلاً الطريقة $f()$ والتي لا تأخذ أي وسيط وتقوم بإرجاع قيمة من نمط *int*. وإذا كان لديك عنصر *a*، عندها يمكنك طلب الطريقة $f()$ من أجل هذا العنصر، كما يمكن كتابة ما يلي:

```
int x = a.f();
```

ويجب أن يكون نمط القيمة التي تم إرجاعها متوافقاً مع نمط *x*.

تسمى عملية استدعاء طريقة بعملية إرسال رسالة إلى عنصر *a* *sending a message to an object*. والرسالة في المثال السابق هي $f()$ أما العنصر فهو *a*.

وتتلخص فكرة البرمجة غرضية التوجه ببساطة على أنها "إرسال رسائل إلى عناصر". أما بالنسبة لقائمة وسطاء طريقة فهي تحدد المعلومات التي يجب تمريرها لهذه الطريقة. هذه المعلومات تأخذ شكل عناصر (مثل أي شيء آخر في جافا). لذلك يجب عليك تحديد



أنماط وأسماء الوسطاء المطلوب تمريرها. ستكون هذه الوسطاء عبارة عن مؤشرات *Handles* إلى العناصر الموافقة.

لنأخذ كمثال طريقة تأخذ وسيطاً من نمط *String*:

```
int storage(String s) {
    return s.length() * 2;
}
```

لاحظ في المثال السابق استخدام الطريقة *length()* التي تُرجع طول عنصر. ويمكنك إرجاع أي نمط ترغب به، لكن عندما لا ترغب بإرجاع أي شيء استخدم نمط *void* المعروف. لنأخذ الأمثلة التالية:

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() {}
```

وعندما تكون القيمة المرجعة من نمط *void*، فإن كلمة المفتاح *return* تستخدم فقط للخروج من الطريقة.

لنبدأ إذا بإنشاء أول برنامج جافا...

هناك العديد من الأمور التي يجب عليك فهمها قبل قيامك بكتابة برنامجك الأول بلغة جافا.

معايير التسميات في جافا...

تعتبر مشكلة التحكم بالتسميات إحدى المشاكل الموجودة في جميع لغات البرمجة. فإذا استخدمت اسماً ما في أحد أجزاء برنامجك، واستخدمت نفس الاسم في جزء آخر، فكيف يمكنك التفريق بين الاسمين ولمنع حدوث أي تضارب؟

تعتبر هذه المشكلة أحد مشاكل لغة *C* لأن البرنامج عادةً يكون عبارة عن بحر من الأسماء التي لا يمكن التحكم بها بشكل فعال. أما في لغة *C++* فإن أسماء الدالات المضمنة ضمن صفٍ لا يمكنها أن تتضارب مع أسماء الدالات المضمنة في صفٍ آخر. لكن تبقى مشكلة

التعارض قائمة لكون لغة ++C تسمح باستخدام المعطيات والدالات العامة. ولحل هذه المشكلة أضافت هذه اللغة ما يسمى بفضاءات الأسماء *namespaces*. أما لغة جافا فلقد تجنبت كل المشاكل السابقة باستخدام تقنية تسميات مختلفة. حيث اعتمدت طريقة التسمية باستخدام اسم مجال إنترنت *Internet Domain Name*. وتقوم هذه الطريقة على مبدأ استخدام اسم مجال إنترنت المعكوس لأن هذا يضمن لك أن يكون هذا الاسم وحيداً.

فإذا افترضنا مثلاً أن اسم مجال إنترنت لديّ هو *MouhibNoukari.com*، فإن مكتبة الألعاب التي أمتلكها ستأخذ الاسم *com.MouhibNoukari.utility.games* في نسختي *Java1.0* و *Java1.1* فإن امتداد المجال *com, edu, org, net* ... إلخ كان يتم أخذه بحرف كبير بشكل افتراضي أي أن التسمية السابقة ستكون على الشكل: *COM.MouhibNoukari.utility.games* أما في نسخة *Java1.2* فتّم اعتماد جميع الأحرف بالحالة الصغيرة.

هذه التقنية المستخدمة في جافا تعني بأن جميع ملفاتك موجودة في فضاءات تسميتها الخاص، وأن كل صف ضمن ملف يأخذ تلقائياً محدّد وحيد *unique identifier*. لذلك لن تكون بحاجة لتعلم أي من تقنيات اللغة الخاصة من أجل حل هذه المشكلة، لأن جافا ستقوم بحلّها عنك.

وإذا احتجت استخدام مكونات أخرى، ماذا أفعل؟

دائماً وعندما تستخدم صفوفاً معروفة مسبقاً *predefined classes* ضمن برنامجك، فإن عليك إخبار المترجم *compiler* عن كيفية إيجادها. ويتم ذلك باستخدام كلمة المفتاح *import*، فمثلاً إذا احتجت استخدام الصف *Vector* الموجود في المكتبة *util* يمكن كتابة:

```
import java.util.Vector;
```

أما عندما تحتاج استخدام جميع صفوف هذه المكتبة فاكُتب:

```
import java.util.*;
```

وماذا تفيدني كلمة المفتاح *static*؟

في الحالة العادية وعندما تقوم بإنشاء صف جديد فأنت تصف كيف ستبدو عناصر هذا الصف وسلوك هذه العناصر. وبالطبع لن تحصل على أي شيء قبل أن تقوم بإنشاء عنصر من هذا الصف باستخدام *new*، حيث يتم إنشاء أماكن تخزين المعطيات بعد ذلك وتصبح الطرق *methods* متاحة.

لكن توجد حالتان تصبح فيهما العملية السابقة غير كافية، الأولى عندما تكون بحاجة إلى جزء من مكان التخزين من أجل جزء خاص للمعطيات بغض النظر عن عدد العناصر التي تم إنشاؤها، أو حتى إذا لم يتم إنشاء أي عنصر. أما الحالة الثانية فهي عندما تحتاج لطريقة غير مرتبطة مع أي عنصر خاص في هذا الصف، أي عندما تحتاج لطريقة يمكن استدعاءها حتى لو لم يتم إنشاء أي عنصر.

في الحالتين السابقتين تستطيع استخدام كلمة المفتاح *static*، والتي تعني بأن الطريقة أو المعطيات غير مرتبطة مع أي عنصر من الصف المحدد. لذلك حتى لو لم تكن قد أنشأت عنصراً من هذا الصف، يمكنك طلب طريقة *static* أو الوصول إلى جزء معطيات *static*. بينما في الحالة العادية *non-static* فعليك إنشاء العنصر أولاً ثم استخدام هذا العنصر للوصول إلى المعطيات أو استخدام الطريقة.

الآن لإنشاء المعطيات أو الطريقة *static*، أضف فقط كلمة المفتاح هذه قبل التعريف. يبين المثال التالي كيفية إنشاء معطيات *static* ووضع قيمة ابتدائية لها:

```
class StaticTest {
    static int i = 47;
}
```

والآن حتى لو قمت بإنشاء عنصرين في الصف *StaticTest*، سيتشارك هذان العنصران بنفس جزء المعطيات *i*:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

عند هذه النقطة، فإن *st1.i* و *st2.i* سيأخذان نفس القيمة 47 لأنهما يدلان على نفس جزء الذاكرة.

وهناك طريقتان للدلالة على متحول *static*. فكما ذكرنا سابقاً يمكنك تسميته باستخدام اسم العنصر مثلاً *s2.i*، أو بالدلالة عليه مباشرة باستخدام اسم الصف الذي ينتمي إليه والذي لا يمكنك استخدامه مع الأعضاء العاديين *non-static members*. لنأخذ مثلاً التعليمة:

```
StaticTest.i++;
```

حيث *++* تقوم بزيادة المتحول قيمة واحدة. عند هذه النقطة فإن *s1.i* و *s2.i* سيأخذان القيمة 48.

نفس الأمر يطبق على الطرق *methods*، حيث يمكنك الدلالة على طريقة ساكنة *static method* إما من خلال عنصر كما في الحالة العادية، أو باستخدام تركيب نحوي خاص *classname.method()*. يمكنك تعريف طريقة ساكنة بشكل مشابه للعبارة:

```
class StaticFun {
    static void incr() { StaticTest.i++; }
}
```

وكما تلاحظ فإن الطريقة *incr()* تقوم بزيادة العداد *i*. ويمكنك طلب هذه الطريقة من خلال اسم العنصر على الشكل:

```
StaticFun sf = new StaticFun();
sf.incr();
```

وباعتبار أن الطريقة *incr()* هي طريقة ساكنة، يمكنك طلب هذه الطريقة مباشرة من خلال الصف الخاص بها:

```
StaticFun.incr();
```

وبينما تغير *static* طريقة إنشاء المعطيات بشكل كامل، فإنها لا تؤثر كثيراً على الطرق.

سأعلمك إذا كتابة أول برنامج بلغة جافا...

أخيراً سنكتب برنامجاً بلغة جافا. وكما تعلم فإن الطريقة الأمثل لتعلم لغة برمجة تكون بالتعرف على كيفية إنشاء برنامج حقيقي ضمن هذه اللغة. والمثال التقليدي الذي يُعطى في البداية هو كتابة برنامج يقوم بإعطاء رسالة "Hello World"، لذلك سنقوم الآن بكتابة برنامج بلغة جافا يقوم بطباعة هذه الرسالة:

```
Class HelloWorld {
    Public static void main(String args[]) {
        System.out.println("Hello, World!");
    }
}
```

وبعد قيامك بترجمة هذا البرنامج، ستكون قادراً على تنفيذه ضمن مفسر جافا *Java Interpreter*. ستجد بأن المترجم يقوم بوضع خرج البرنامج في ملف بالاسم *.HelloWorld.class* ومن أجل تنفيذ هذا البرنامج، اكتب العبارة:

```
java HelloWorld
```

في سطر الأوامر، وستلاحظ ظهور الرسالة "Hello, World" على شاشتك.

تهانينا! فلقد بدأت الآن بالدخول إلى عالم جافا.

ملاحظة: في بعض بيئات البرمجة قد تلاحظ ظهور البرنامج على الشاشة ثم إغلاقه قبل أن يتسنى لك فرصة رؤية النتيجة. في هذه الحالة تستطيع إضافة أسطر الترميز التالية في نهاية الطريقة *main()* لإيقاف الخرج:

```
try {
    Thread.currentThread().sleep(5 * 1000);
} catch (InterruptedException e) {}
}
```

هذا سيؤدي إلى إيقاف الخرج لمدة خمس ثوانٍ. وسنشرح في الفصول القادمة المفاهيم المنضمة في هذين السطرين لذلك لا تقلق!!

ملاحظة: يمكن وضع عدة أسطر كأسطر تعليق بوضعها بين */* ... */* كما في لغة *C*.

تساعدك لغة جافا حتى على توليد توثيق لبرامجك...

من الأمور المتميزة التي أتت بها لغة جافا القدرة على توليد البرامج *Program Documentation* التي تقوم بكتابتها. ولقد كانت مشكلة صيانة التوثيق من أكبر المشاكل التي تواجه المبرمجين. فإذا كان توثيق أي برنامج منفصلاً عن ترميز هذا البرنامج فستجد صعوبة في تغيير التوثيق في كل مرة تقوم فيها بتغيير أي جزء من الترميز.

لذلك فإن الحل ببساطة يكون بربط التوثيق بالترميز وذلك بوضعها في نفس الملف، وهو ما تقوم به بلغة جافا والتي تستخدم الأداة *javadoc* للحصول على التعليقات. وتستخدم جافا بعض تقنيات مترجم اللغة للبحث عن بعض علامات *tags* التعليق الخاصة وتضعها في البرنامج. وهي لا تقوم فقط ب جلب المعلومات المحددة بهذه العلامات وإنما تستخرج اسم الصف أو الطريقة والمرتبطة بالتعليق. ونتيجة *javadoc* فهي عبارة عن ملف *HTML* تستطيع رؤيته باستخدام مستعرض *Web* الموجود لديك.

لنتعرف الآن على تركيبة التوثيق...

تتواجد جميع تعليمات *javadoc* بين تعليقات */*** فقط والتي تنتهي بـ **/*. وتوجد طريقتان أساسيتان لاستخدام *javadoc*: إما *HTML* *embed* أو *doc tags*.

وتبدأ تعليمات *doc tags* بالرمز *@* ويتم وضعها في بداية سطر التعليق. توجد ثلاثة أنماط لتوثيق التعليقات ترتبط بعنصر التعليق الذي يسبقها، فهي إما الصف *class* أو المتحول *variable* أو الطريقة *method* وذلك كما في المثال التالي:

```
/** A class comment */
public class docTest {
```



```
/** A variable comment */
public int i;
/** A method comment */
public void f() {}
}
```

لاحظ أن *javadoc* تعالج توثيق التعليق للأعضاء من نمط *public* أو *protected* فقط، أما بالنسبة للأعضاء *private* أو *friendly* فيتم تجاهلهم، والسبب في ذلك هو أن الأعضاء *public* أو *protected* هي المتاحة خارج الملف فقط.

خرج الترميز السابق عبارة عن ملف *HTML* الذي يأخذ نفس التنسيق القياسي لبقية توثيق جافا وهو ما سيريح المستخدمين كثيراً لأنه سيمنحهم من التنقل بين صفوفهم بسهولة.

ما الذي تعنيه بـ *HTML* المضمنة؟

تقوم الأداة *javadoc* بتمرير تعليمات *HTML* إلى وثيقة *HTML* المولدة. مما يسمح لك بترميز التنسيق كما يلي:

```
/**
 * <pre>
 * System.out.println(new Date());
 * </pre>
 */
```

تستطيع أيضاً استخدام *HTML* كما ترغب في أي وثيقة *Web* أخرى لتنسيق النص العادي:

```
/**
 * You can <em>even</em> insert a list:
 * <ol>
 * <li> Item one
 * <li> Item two
 * <li> Item three
 * </ol>
 */
```

لاحظ أنه ضمن تعليق التوثيق، يقوم `javadoc` بإلغاء "/" من بداية كل سطر. ويجب ألا تقوم باستخدام ترويسات مثل `<h1>` أو `<hr>` كتعليمات `HTML` مضمنة لأن `javadoc` يقوم بإدراج ترويساته الخاصة وستتضارب عندها مع الترويسات السابقة.

تفليك علامة `@see` للدلالة على صفوف أخرى...

يمكن لجميع الأنماط الثلاثة من توثيق التعليق أن تتضمن علامات `@see` والتي تسمح لك بالرجوع إلى توثيق صفوف أخرى. وسيقوم `javadoc` بتوليد `HTML` مع علامات `@see` وإنشاء ارتباط فائق `Hyperlink` للتوثيقات الأخرى. وهي تأخذ الشكل:

```
@see classname
```

```
@see fully-qualified-classname
```

```
@see fully-qualified-classname#method-name
```

كل من التعليمات السابقة تقوم بإضافة مدخل ارتباط فائق "See Also" إلى التوثيق المولد. ولن يتحقق `javadoc` من هذه الارتباطات الفائقة للتأكد من أنها محققة.

علامات توثيق الصف...

يمكن لتوثيق الصف أن يحتوي على علامات تتعلق بمعلومات النسخة `version` `information` واسم المؤلف. ويمكن لتوثيق الصف أن يستخدم أيضاً للواجهات `interfaces` (كما سنرى في الفصول القادمة). بالنسبة لعلامات معلومات النسخة فتأخذ الشكل:

```
@version version-information
```

أما بالنسبة لعلامة اسم المؤلف فتأخذ الشكل:

```
@author author-information
```

علامات توثيق المتحولات...

يمكن لتوثيق المتحولات `Variable Documentation` أن يحتوي فقط على `HTML` مضمنة ومؤشرات `@see`.



علامات توثيق الطرق...

إضافةً إلى التوثيق المضمّن ومؤشرات `@see`، يمكن للطرق أن تسمح بعلامات لتوثيق الوسائط `parameters` وقيم الإرجاع `return values` والاستثناءات `exceptions`.

أما علامة الوسيط فتأخذ الشكل:

```
@param parameter- name description
```

وتأخذ علامة قيم الإرجاع الشكل:

```
@return description
```

وتأخذ علامة الاستثناءات الشكل:

```
@exception fully-qualified-class-name
description
```

ملاحظة: يفضل عند تسمية الصفوف كتابة الحرف الأول بحرف كبير، كذلك عند استخدام اسم مكون من عدة كلمات قم بتكبير الحرف الأول (ولا تقم باستخدام `underscore` للفصل بين الكلمات) كما في الشكل:

```
class AllTheColorsOfTheRainbow { // ...
```

وبالنسبة لأي شيء آخر مثل الطرق `methods` والحقول `fields` وأسماء مؤشرات العناصر `object handles` فتأخذ نفس الافتراضات السابقة عدا أن الحروف الأولى من المحدّد سيكون بحرف صغير، مثلاً:

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}
```




في لغة جافا معالجة العناصر والمعطيات باستخدام المعاملات

تستطيع Operators. وتستطيع تحديد خياراتك من خلال تعليمات التحكم بالتنفيذ.

وعلى اعتبار أن توريث لغة جافا قد تم من لغة C++، لذلك ستكون أغلب المعاملات والتعليمات الموجودة فيها معروفة بالنسبة لمبرمجي C و C++. بالطبع لقد أضافت جافا بعض التسهيلات والتحسينات.

سلسلة الرضا للمعلومات

المعاملات في جافا...

يوضح الجدول التالي المعاملات الأساسية المستخدمة في لغة جافا:

+, -, *, /, %	المعاملات الحسابية
++, --	معاملات الزيادة والنقصان
==, !=, >, <, >=, <=	المعاملات العلائقية
&&, , !	المعاملات المنطقية
&, ^, >>, <<, >>>, <<<	معاملات Bitwise
+	معامل الدمج لسلاسل المحارف
=	معامل النسب

سنوضح من خلال الأمثلة التالية كيفية استخدام هذه المعاملات:

مثال ١: عن كيفية استخدام معامل النسب في جافا:

```
//: Assignment.java
// Assignment with objects is a bit tricky
package c03;
class Number {
    int i;
}
public class Assignment {
    public static void main(String[] args) {
        Number n1 = new Number();
        Number n2 = new Number();
        n1.i = 9;
        n2.i = 47;
        System.out.println("1: n1.i: " + n1.i +
            ", n2.i: " + n2.i);
        n1 = n2;
        System.out.println("2: n1.i: " + n1.i +
            ", n2.i: " + n2.i);
        n1.i = 27;
        System.out.println("3: n1.i: " + n1.i +
            ", n2.i: " + n2.i);
    }
}
```



```

    }
} ///:~

```

كنتيجة لتنفيذ البرنامج السابق ستحصل على ما يلي:

```

1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27

```

وتلاحظ ظهور نفس القيمة للعنصرين $n1$ و $n2$ لأنهما يؤشران عند هذه النقطة على نفس العنصر.

تسمى هذه الظاهرة بالترادف *Aliasing* والتي تعتبر الطريقة الأساسية التي نتعامل بها جافا مع العناصر.

وتستطيع استخدام الترادف عند طلب طريقة كما في المثال التالي:

```

//: PassObject.java
// Passing objects to methods can be a bit
tricky
class Letter {
    char c;
}
public class PassObject {
    static void f(Letter y) {
        y.c = 'z';
    }
    public static void main(String[] args) {
        Letter x = new Letter();
        x.c = 'a';
        System.out.println("1: x.c: " + x.c);
        f(x);
        System.out.println("2: x.c: " + x.c);
    }
} ///:~

```

حيث ستحصل على النتيجة التالية عند تنفيذ هذا البرنامج:

```

1: x.c: a
2: x.c: z

```

مثال ٢: يبين كيفية استخدام المعاملات الحسابية:

```

//: MathOps.java

```

سلسلة الرضا للمعلومات

```
// Demonstrates the mathematical operators
import java.util.*;
public class MathOps {
    // Create a shorthand to save typing:
    static void prt(String s) {
        System.out.println(s);
    }
    // shorthand to print a string and an int:
    static void pInt(String s, int i) {
        prt(s + " = " + i);
    }
    // shorthand to print a string and a float:
    static void pFlt(String s, float f) {
        prt(s + " = " + f);
    }
    public static void main(String[] args) {
        // Create a random number generator,
        // seeds with current time by default:
        Random rand = new Random();
        int i, j, k;
        // '%' limits maximum value to 99:
        j = rand.nextInt() % 100;
        k = rand.nextInt() % 100;
        pInt("j",j); pInt("k",k);
        i = j + k; pInt("j + k", i);
        i = j - k; pInt("j - k", i);
        i = k / j; pInt("k / j", i);
        i = k * j; pInt("k * j", i);
        i = k % j; pInt("k % j", i);
        j %= k; pInt("j %= k", j);
        // Floating-point number tests:
        float u,v,w; // applies to doubles, too
        v = rand.nextFloat();
        w = rand.nextFloat();
        pFlt("v", v); pFlt("w", w);
        u = v + w; pFlt("v + w", u);
        u = v - w; pFlt("v - w", u);
        u = v * w; pFlt("v * w", u);
    }
}
```




```

    u = v / w; pFlt("v / w", u);
    // the following also works for
    // char, byte, short, int, long,
    // and double:
    u += v; pFlt("u += v", u);
    u -= v; pFlt("u -= v", u);
    u *= v; pFlt("u *= v", u);
    u /= v; pFlt("u /= v", u);
}
} ///:~

```

مثال ٣: يوضح كيفية استخدام معاملات الزيادة والنقصان:

```

//: AutoInc.java
// Demonstrates the ++ and -- operators
public class AutoInc {
    public static void main(String[] args) {
        int i = 1;
        prt("i : " + i);
        prt("++i : " + ++i); // Pre-increment
        prt("i++ : " + i++); // Post-increment
        prt("i : " + i);
        prt("--i : " + --i); // Pre-decrement
        prt("i-- : " + i--); // Post-decrement
        prt("i : " + i);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~

```

وعند تنفيذ البرنامج السابق ستحصل على النتيجة التالية:

```

i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1

```

مثال ٤: يوضح المثال التالي كيفية استخدام معاملات المقارنة العلائقية:

```
//: Equivalence.java
public class Equivalence {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);
        System.out.println(n1 != n2);
    }
} ///:~
```

وستحصل بالطبع على النتيجة التالية:

```
True
False
```

أما إذا أردت إجراء المقارنة بين محتوى عنصرين سواء أكانا أوليين أم لا، تستطيع في هذه الحالة استخدام الطريقة `equals()` كما في المثال التالي:

```
//: EqualsMethod.java
public class EqualsMethod {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1.equals(n2));
    }
} ///:~
```

مثال ٥: يوضح كيفية استخدام المعاملات المنطقية:

```
//: Bool.java
// Relational and logical operators
import java.util.*;
public class Bool {
    public static void main(String[] args) {
        Random rand = new Random();
        int i = rand.nextInt() % 100;
        int j = rand.nextInt() % 100;
        prt("i = " + i);
        prt("j = " + j);
        prt("i > j is " + (i > j));
    }
}
```

```

    prt("i < j is " + (i < j));
    prt("i >= j is " + (i >= j));
    prt("i <= j is " + (i <= j));
    prt("i == j is " + (i == j));
    prt("i != j is " + (i != j));
    // Treating an int as a boolean is
    // not legal Java
    ///! prt("i && j is " + (i && j));
    ///! prt("i || j is " + (i || j));
    ///! prt("!i is " + !i);
    prt("(i < 10) && (j < 10) is "
        + ((i < 10) && (j < 10)) );
    prt("(i < 10) || (j < 10) is "
        + ((i < 10) || (j < 10)) );
}
static void prt(String s) {
    System.out.println(s);
}
} ///:~

```

وستحصل هنا على النتيجة التالية:

```

i = 85
j = 4
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is true

```

وماهي أفضلّيات المعاملات في لغة جافا؟

يوضح الجدول التالي أفضلّيات المعاملات في جافا:

<i>Mnemonic</i>	<i>Operator type</i>	<i>Operators</i>
<i>Ulcer</i>	<i>Unary</i>	$+ - ++ - [[rest...]]$
<i>Addicts</i>	<i>Arithmetic (and shift)</i>	$* / \% + - << >>$
<i>Really</i>	<i>Relational</i>	$> < >= <= == !=$
<i>Like</i>	<i>Logical (and bitwise)</i>	$\&\& \& ^$
<i>C</i>	<i>Conditional (ternary)</i>	$A > B ? X : Y$
<i>A Lot</i>	<i>Assignment</i>	$=$ (and compound assignment like $*=$)

التعليمات الأساسية في جافا...

لغة جافا كغيرها من لغات البرمجة تمتلك مجموعة من التعليمات الأساسية التي تساعد على التحكم بعملية التنفيذ *Execution Control*.

تعليمية الشرط *if-else*...

نأخذ هذه التعليمية الشكل العام:

```
if(Boolean-expression)
    statement
```

أو:

```
if(Boolean-expression)
    statement
else
    statement
```

كمثال على استخدام التعليمية السابقة نأخذ:

```
static int test(int testval) {
```



```

int result = 0;
if(testval > target)
    result = -1;
else if(testval < target)
    result = +1;
else
    result = 0; // match
return result;
}

```

ولتعليمية `return` فائدتان: الأولى تحديد القيمة التي سترجعها الطريقة، أما الثانية فهي إرجاع هذه القيمة مباشرة. ويمكن إعادة كتابة المثال السابق للاستفادة من إمكانيات `:return`

```

static int test2(int testval) {
    if(testval > target)
        return -1;
    if(testval < target)
        return +1;
    return 0; // match
}

```

ملاحظة: يمكن استخدام معاملاً خاصاً يقوم بتنفيذ تعليمية الشرط `if-else` وذلك على الشكل:

`boolean-exp ? value0 : value1`

حيث يتم اختبار الشرط `boolean-exp` فإذا تحقق استخدم `value0` وإلا فاستخدم `value1`، كمثال على ذلك نأخذ:

```

static int ternary(int i) {
    return i < 10 ? i * 100 : i * 10;
}

```

هذا المثال يوافق تماماً كتابة الترميز التالي:

```

static int alternative(int i) {
    if (i < 10)
        return i * 100;
    return i * 10;
}

```

تعليمات التكرار في جافا؟

يوجد العديد من تعليمات التكرار في لغة جافا أهمها:

✓ تعليمة **While**: والتي تأخذ الشكل العام:

```
while(Boolean-expression)
    statement
```

كمثال على التعليمة السابقة:

```
//: WhileTest.java
// Demonstrates the while loop
public class WhileTest {
    public static void main(String[] args) {
        double r = 0;
        while(r < 0.99d)
            r = Math.random();
        System.out.println(r);
    }
} ///:~
```

✓ تعليمة **Do-While**: تأخذ هذه التعليمة الشكل العام:

```
do
    statement
while(Boolean-expression);
```

✓ تعليمة **for**: تأخذ هذه التعليمة الشكل العام:

```
for(initialization; Boolean-expression; step)
    statement
```

وكمثال على استخدام هذه التعليمة نكتب:

```
//: ListCharacters.java
// Demonstrates "for" loop by listing
// all the ASCII characters.
public class ListCharacters {
    public static void main(String[] args) {
        for( char c = 0; c < 128; c++)
            if (c != 26 ) // ANSI Clear screen
                System.out.println(
```



```

        "value: " + (int)c +
        " character: " + c);
    }
} ///:~

```

ويمكن استخدام عدة متحولات ضمن تعليمة *for*، لكن يجب أن تكون جميعها من نفس النمط، مثلاً:

```

for(int i = 0, j = 1;
    i < 10 && j != 11;
    i++, j++)
/* body of for loop */;

```

لاحظ أنه من أجل الفصل بين المتحولات يجب استخدام الفاصلة "،". الآن، وضمن جسم أي تعليمة تكرار، بإمكانك التحكم بتدفق الحلقة باستخدام تعليمتي *Break* و *Continue*. حيث تقوم تعليمة *Break* بالخروج من الحلقة دون تنفيذ بقية التعليمات فيها. أما تعليمة *Continue* فتوقف التكرار الحالي وتعود إلى تنفيذ بداية الحلقة وبدء تكرار جديد.

يوضح المثال التالي كيفية استخدام التعليمتين السابقتين:

```

//: BreakAndContinue.java
// Demonstrates break and continue keywords
public class BreakAndContinue {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            if(i == 74) break; // Out of for loop
            if(i % 9 != 0) continue; // Next
            iteration
            System.out.println(i);
        }
    }
}
int i = 0;
// An "infinite loop":
while(true) {
    i++;
    int j = i * 27;
    if(j == 1269) break; // Out of loop
    if(i % 10 != 0) continue; // Top of loop
    System.out.println(i);
}

```

```

    }
}
} ///:~

```

تأكد من ظهور النتيجة التالية عند تنفيذ هذا البرنامج:

```

0
9
18
27
36
45
54
63
72
10
20
30
40

```

ولقد ظهرت النتيجة 0 لأن 9% 0 تعطي 0.

حتى أنه يمكنك استخدام تعليمة goto المتخلفة!!!

لقد أتاحت لغة جافا استخدام هذه التعليمة القديمة قدم لغات البرمجة، وهي هنا تأخذ الشكل:

```

label1:
outer-iteration {
inner-iteration {
//...
break; // 1
//...
continue; // 2
//...
continue label1; // 3
//...
break label1; // 4
}
}

```

كمثال على هذه التعليمة:



```

//: LabeledFor.java
// Java's "labeled for loop"
public class LabeledFor {
    public static void main(String[] args) {
        int i = 0;
        outer: // Can't have statements here
        for(;; true ;) { // infinite loop
            inner: // Can't have statements here
            for(;; i < 10; i++) {
                prt("i = " + i);
                if(i == 2) {
                    prt("continue");
                    continue;
                }
                if(i == 3) {
                    prt("break");
                    i++; // Otherwise i never
                        // gets incremented.
                    break;
                }
                if(i == 7) {
                    prt("continue outer");
                    i++; // Otherwise i never
                        // gets incremented.
                    continue outer;
                }
                if(i == 8) {
                    prt("break outer");
                    break outer;
                }
                for(int k = 0; k < 5; k++) {
                    if(k == 3) {
                        prt("continue inner");
                        continue inner;
                    }
                }
            }
        }
    }
}

```

```
// Can't break or continue
// to labels here
}
static void prt(String s) {
    System.out.println(s);
}
} ///:~
```

عند تنفيذ هذا البرنامج يجب أن تظهر لديك النتيجة التالية:

```
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
```

تعليلة الاختيار Switch

كما نعلم فإن تعليلة switch تقوم بالاختيار بين عدة أجزاء من الترميز وذلك بالاعتماد على قيمة معينة، وهي تأخذ الشكل:

```
switch(integral-selector) {
    case integral-value1 : statement; break;
    case integral-value2 : statement; break;
    case integral-value3 : statement; break;
    case integral-value4 : statement; break;
```

```

case integral-value5 : statement; break;
// ...
default: statement;
}

```

كمثال على هذه التعليمة:

```

//: VowelsAndConsonants.java
// Demonstrates the switch statement
public class VowelsAndConsonants {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            char c = (char) (Math.random() *
26 + 'a');
            System.out.print(c + ": ");
            switch(c) {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                    System.out.println("vowel")
                    ;
                    break;
                case 'y':
                case 'w':
                    System.out.println(
                        "Sometimes a vowel");
                    break;
                default:
                    System.out.println("consonant");
            }
        }
    }
} ///:~

```




أهم العلل والمشاكل التي كانت تواجه مبرمجي لغة C نسيانهم وضع قيمة ابتدائية لمتحول، وبشكل خاص عندما لا يعرفون كيفية وضع قيمة ابتدائية من مكتبة *Library Component*. إضافة إلى ذلك المشاكل المتعلقة بمسح العناصر من الذاكرة عند الانتهاء من استخدامها، مما يبقي المصادر *Resources* المستخدمة من قبل هذه العناصر محجوزة، وقد تصل إلى مرحلة لا تعد تتوفر لديك فيها أية مصادر.

ولقد أضافت لغة ++C مفهوم الباني *Constructor*، وهو عبارة عن طريقة خاصّة تُطلب تلقائياً عند إنشاء أي عنصر. ولقد اعتمدت جافا أيضاً مفهوم الباني وأضافت إليه

مفهوم مجمع النفايات *Garbage Collector*، والذي يقوم بتحرير مصادر الذاكرة تلقائياً عند الانتهاء من استخدامها.
سنقوم في هذا الفصل بتحديد كيفية القيام بعمليات تحديد القيمة الابتدائية *Initialization* ومسح العناصر *Cleanup* في لغة جافا.

عند استخدامك للبيانات *Constructors* ستتخلص من مشكلة تحديد القيمة الابتدائية...

يمكنك أن تتخيل إنشاء طريقة اسمها *initialize()* تقوم باستخدامها عند إنشاء أي عنصر، ويجب طلبها دوماً قبل استخدام هذا العنصر. هذا يعني أن على المبرمج تذكر استخدام هذه الطريقة بشكل دائم عند الحاجة لإنشاء عنصر جديد.

جافا استطاعت حل هذه المشكلة باعتمادها على طريقة خاصة هي الباني *Constructor*. فإذا امتلك صف ما هذا الباني، يستطيع وبشكل تلقائي استدعاءه عند إنشاء عنصر حتى قبل أن يتمكن المبرمج من إجراء أي عملية عليه، مما يضمن حل مشكلة تحديد القيمة الابتدائية.

لكن كيف نستطيع اختيار اسم للباني؟ فقد يتضارب الاسم المختار مع اسم أي عضو من أعضاء الصف. إضافة إلى ذلك، يجب على المترجم *Compiler* معرفة أي طريقة يقوم باستدعائها عند طلب الباني.

الحل الذي استخدمته لغة *C++* يبدو أسهل وأكثر منطقية لذلك فقد اعتمدته لغة جافا، والذي يقوم على أخذ اسم الباني كاسم الصف، مما يعني بأن هذا الباني سيتم استدعاؤه تلقائياً. يبين المثال التالي كيفية إنشاء صف بسيط واستخدام بان لهذا الصف:

```
//: SimpleConstructor.java
// Demonstration of a simple constructor
package c05;
class Rock {
    Rock() { // This is the constructor
```



```

        System.out.println("Creating Rock");
    }
}
public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
} ///:~

```

الآن عندما نقوم بإنشاء العنصر باستخدام التعليمة:

```
new Rock();
```

يتم حجز مكان لهذا العنصر ومن ثم طلب الباني. وسيتم تحديد قيمة ابتدائية له قبل استخدام هذا العنصر.

لاحظ هنا أن اسم الباني مشابه تماماً لاسم الصف.

وكأي طريقة، يمكن أن يحتوي الباني على وسطاء تسمح له بتحديد كيفية إنشاء عنصر. سنقوم بتعديل المثال السابق بحيث يتم استخدام وسطاء ضمن الباني على الشكل التالي:

```

class Rock {
    Rock(int i) {
        System.out.println(
            "Creating Rock number " + i);
    }
}
public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock(i);
    }
}

```

هذه الطريقة تسمح لك بتحديد وسطاء من أجل القيمة الابتدائية لعنصر. فإذا كان لدينا كمثال الصف *Tree* والذي يمتلك بان يأخذ عدداً طبيعياً كوسيط يحدّد ارتفاع الشجرة، عندها يمكنك إنشاء عنصر *Tree* كما في الشكل:

```
Tree t = new Tree(12); // 12-foot tree
```

فإذا كان `Tree(int)` الباني الوحيد، فلن يسمح لك المترجم بإنشاء عنصر `Tree` بطريقة أخرى.

تسمح لك البانيات بالتخلص من عدة مشاكل الصفوف، كما تجعل من السهل قراءة الترميز الموافق. وهي تعتبر طرقا غير عادية كونها لا تمتلك قيمة مرجعة `return value` (هذا بالطبع يختلف عن إرجاع القيمة `Void`).

تحميل الطرق *Method Overloading*

تستخدم أغلب لغات البرمجة (و `C` بالتحديد) محددا وحيدا لكل دالة. لذا لن يكون بإمكانك استخدام دالة `print()` لطباعة الأعداد الطبيعية مثلا، ودالة أخرى بنفس الاسم `print()` لطباعة الأعداد الحقيقية.

أما في لغة جافا فالأمر مختلف تماما والسبب في ذلك هو أنك في كثير من الأحيان تحتاج إلى استخدام الباني من أجل إنشاء العناصر بأكثر من شكل. لهذا يجب أن تستخدم نفس اسم طريقة الباني لإنشاء العناصر بجميع الأشكال المطلوبة، وهو ما نسميه بتحميل الطرق *Method Overloading*.

لنفترض مثلا بأنك تريد بناء صف يستطيع تحديد قيمته الابتدائية، إما باستخدام الطريقة القياسية، أو بقراءة المعلومات من ملف. في هذه الحالة تحتاج إلى استخدام `String` كوسيط. وباعتبار أن اسم الباني يجب أن يشبه اسم الصف تماما، لذلك في هذه الحالة نحن بحاجة إلى تحميل الطرق من أجل السماح باستخدام نفس اسم الطريقة بأنماط وسطاء مختلفة.

وعلى الرغم من أن استخدام تحميل الطرق ضروري عند استخدام البانيات، إلا أنه يستخدم أيضا للطرق العادية. يوضح المثال التالي كيفية استخدام تحميل الطرق للبانيات وللطرق العادية:

```
//: Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
```



```

import java.util.*;
class Tree {
    int height;
    Tree() {
        prt("Planting a seedling");
        height = 0;
    }
    Tree(int i) {
        prt("Creating new Tree that is "
            + i + " feet tall");
        height = i;
    }
    void info() {
        prt("Tree is " + height
            + " feet tall");
    }
    void info(String s) {
        prt(s + ": Tree is "
            + height + " feet tall");
    }
    static void prt(String s) {
        System.out.println(s);
    }
}
public class Overloading {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("overloaded method");
        }
        // Overloaded constructor:
        new Tree();
    }
} ///:~

```

لكن كيف تستطيع جافا التمييز بين الطرق المحملة؟

الطريقة بسيطة جدا" وتعتمد على أن كل طريقة محملة تأخذ قائمة وحيدة من أنماط الوسطاء، وأظن بأنه لا توجد طريقة أخرى!!؟
والأهم من ذلك هو أن ترتيب الوسطاء كاف للتمييز بين طريقتين، قم بتطبيق المثال التالي للتأكد مما أقوله لك:

```
//: OverloadingOrder.java
// Overloading based on the order of
// the arguments.
public class OverloadingOrder {
    static void print(String s, int i) {
        System.out.println(
            "String: " + s +
            ", int: " + i);
    }
    static void print(int i, String s) {
        System.out.println(
            "int: " + i +
            ", String: " + s);
    }
    public static void main(String[] args) {
        print("String first", 11);
        print(99, "Int first");
    }
} ///:~
```

التحميل باستخدام الأنماط الأولية...

يوضح المثال التالي كيفية استخدام الأنماط الأولية مع الطرق المحملة:

```
//: PrimitiveOverloading.java
// Promotion of primitives and overloading
public class PrimitiveOverloading {
    // boolean can't be automatically converted
    static void prt(String s) {
        System.out.println(s);
    }
}
```



```

}
void f1(char x) { prt("f1(char)"); }
void f1(byte x) { prt("f1(byte)"); }
void f1(short x) { prt("f1(short)"); }
void f1(int x) { prt("f1(int)"); }
void f1(long x) { prt("f1(long)"); }
void f1(float x) { prt("f1(float)"); }
void f1(double x) { prt("f1(double)"); }
void f2(byte x) { prt("f2(byte)"); }
void f2(short x) { prt("f2(short)"); }
void f2(int x) { prt("f2(int)"); }
void f2(long x) { prt("f2(long)"); }
void f2(float x) { prt("f2(float)"); }
void f2(double x) { prt("f2(double)"); }
void f3(short x) { prt("f3(short)"); }
void f3(int x) { prt("f3(int)"); }
void f3(long x) { prt("f3(long)"); }
void f3(float x) { prt("f3(float)"); }
void f3(double x) { prt("f3(double)"); }
void f4(int x) { prt("f4(int)"); }
void f4(long x) { prt("f4(long)"); }
void f4(float x) { prt("f4(float)"); }
void f4(double x) { prt("f4(double)"); }
void f5(long x) { prt("f5(long)"); }
void f5(float x) { prt("f5(float)"); }
void f5(double x) { prt("f5(double)"); }
void f6(float x) { prt("f6(float)"); }
void f6(double x) { prt("f6(double)"); }
void f7(double x) { prt("f7(double)"); }
void testConstVal() {
prt("Testing with 5");
f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
void testChar() {
    char x = 'x';
    prt("char argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testByte() {

```

```

byte x = 0;
prt("byte argument:");
f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testShort() {
    short x = 0;
    prt("short argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testInt() {
    int x = 0;
    prt("int argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testLong() {
    long x = 0;
    prt("long argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testFloat() {
    float x = 0;
    prt("float argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testDouble() {
    double x = 0;
    prt("double argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
public static void main(String[] args) {
    PrimitiveOverloading p =
    new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
}

```



```
p.testFloat();
p.testDouble();
}
} ///:~
```

لقد نسيت إنشاء باني ضمن الصف، فماذا أفعل؟

لا تقلق لأن مترجم جافا سيقوم تلقائياً بإنشاء الباني الافتراضي عنك. لنأخذ المثال التالي:

```
//: DefaultConstructor.java
class Bird {
    int i;
}
public class DefaultConstructor {
    public static void main(String[] args) {
        Bird nc = new Bird(); // default!
    }
} ///:~
```

في البرنامج السابق سيقوم السطر:

```
new Bird();
```

بإنشاء العنصر واستدعاء الباني الافتراضي حتى لو لم يتم تعريفه.

ما هي الفائدة من كلمة المفتاح *this*؟

لنفترض لدينا عنصرين من نفس النمط هما a و b ، وقد ترغب بمعرفة كيف يمكنك استدعاء الطريقة $f()$ لهذين العنصرين:

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

فإذا كانت لدينا طريقة واحدة فقط بالاسم $f()$ ، فكيف بإمكانها معرفة فيما إذا تم استدعاؤها للعنصر a أم للعنصر b ؟

قد تكون الطريقة غير سهلة إلا أن عليك البقاء ضمن مفهوم البرمجة غرضية التوجه، أي أنك ترغب بكتابة شيء يشبه:

```
Banana.f(a,1);
```

```
Banana.f(b,2);
```

طبعا التعليمتان السابقتان غير صحيحتين ولن يقبل بهما المترجم. لنفترض الآن أنك ضمن طريقة ترغب بالحصول على المؤشر للعنصر الحالي. وباعتبار أن تمرير المؤشر يتم بشكل سري من قبل المترجم، لذلك لن يكون هناك محدد لهذا المؤشر. من هنا تأتي أهمية استخدام كلمة المفتاح *this* والتي يمكن استخدامها فقط ضمن طريقة من أجل الوصول إلى المؤشر للعنصر الذي قام باستدعاء هذه الطريقة. تذكر أنك عندما تقوم باستدعاء طريقة موجودة لديك من خلال طريقة أخرى ضمن الصف، فلن تكون بحاجة لاستخدام *this*. ويتم استخدام المؤشر *this* تلقائيا في الطريقة الأخرى. لذلك تستطيع القول:

```
class Apricot {
    void pick() { /* ... */
    void pit() { pick(); /* ... */
}
```

الآن داخل *pit()* تستطيع القول *this.pick()* لكن لا حاجة لذلك، لأن المترجم يقوم به تلقائيا.

تستخدم إذا *this* فقط في الحالات الخاصة التي نحتاج فيها وبشكل خارجي إلى التعامل مع مؤشر العنصر الحالي. وهو غالبا ما يستخدم في تعليمة *return* كما في المثال التالي:

```
//: Leaf.java
// Simple use of the "this" keyword
public class Leaf {
    private int i = 0;
    Leaf increment() {
        i++;
        return this;
    }
    void print() {
        System.out.println("i = " + i);
    }
    public static void main(String[] args) {
        Leaf x = new Leaf();
```



```
x.increment().increment().increment().print
();
}
} ///:~
```

يمكنك أيضا استدعاء باني من خلال بان آخر!!؟

عندما تقوم بكتابة عدة بانايات لصف واحد، قد تحتاج في بعض الأحيان إلى طلب أحدها من خلال بان آخر لتجنب تكرار كتابة الترميز. تستطيع الآن القيام بذلك باستخدام *this*، كيف!!؟ سنرى ذلك معا...

عندما تستخدم عادة *this* فإنك تدل على العنصر الحالي أو كما ذكرنا مرارا وتكرارا تعطي مؤشرا على العنصر الحالي.

أما في البانايات فقد يختلف استخدام *this* قليلا عندما تقوم بإعطائه قائمة وسطاء. حيث يتم استدعاء الباني الذي يتوافق مع قائمة الوسطاء بشكل خارجي. كمثال على ذلك:

```
//: Flower.java
// Calling constructors with "this"
public class Flower {
    private int petalCount = 0;
    private String s = new String("null");
    Flower(int petals) {
        petalCount = petals;
        System.out.println(
            "Constructor w/ int arg only, petalCount= "
            + petalCount);
    }
    Flower(String ss) {
        System.out.println(
            "Constructor w/ String arg only, s=" + ss);
        s = ss;
    }
    Flower(String s, int petals) {
        this(petals);
        //! this(s); // Can't call two!
        this.s = s; // Another use of "this"
```

```

System.out.println("String & int args");
}
Flower() {
    this("hi", 47);
    System.out.println(
        "default constructor (no args)");
}
void print() {
    ///! this(11); // Not inside non-
    constructor!
    System.out.println(
        "petalCount = " + petalCount + " s = " + s);
}
public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
}
} ///:~

```

من خلال الباني `Flower(String s, int petals)` نجد مع أنه بالإمكان استدعاء بان واحد باستخدام `this` إلا أنك لن تستطيع استدعاء بانين اثنين. بالإضافة إلى ذلك فإن عليك استدعاء الباني وقبل أي شيء وإلا فستحصل على رسالة خطأ من المترجم.

مجمع النفايات *Garbage Collector*

يركز معظم المبرمجين جل اهتمامهم على عملية تحديد القيم الابتدائية لعناصرهم، و ينسون بعد ذلك أن يقوموا بالتخلص من هذه العناصر بعد الانتهاء منها، وهذا ما نسميه بعملية التنظيف *cleanup*.

كما ذكرنا سابقا فإن جافا تمتلك أداة فعالة للتخلص تلقائيا من العناصر وهي مجمع النفايات *Garbage Collector*، حيث تقوم هذه الأداة بتحرير مصادر الذاكرة المحجوزة من قبل العناصر التي لم نعد بحاجة إلى استخدامها.



لكن لنأخذ حالة خاصة جدا وغير عادية، ولنفترض أن عنصرك يقوم بحجز ذاكرة خاصة دون استخدام *new*. وعلى اعتبار أن مجمع النفايات يعرف كيفية تحرير الذاكرة المحجوزة باستخدام *new* فقط، لذلك لن يتمكن من معرفة كيفية تحرير هذه الذاكرة الخاصة.

من أجل معالجة هذه الحالة، تستخدم جافا طريقة خاصة *finalize()* يمكنك تعريفها ضمن أي صف.

سأشرح لك الآن كيفية العمل مع الطريقة السابقة...

عندما يكون مجمع النفايات جاهزا لتحرير الذاكرة المحجوزة لعنصر، سيستدعي أولا الطريقة *finalize()*، وسيقوم بتحرير الذاكرة المحجوزة للعنصر فقط عند المرور الثاني لمجمع النفايات. هذا بالطبع سيعطيك القدرة على إجراء بعض عمليات التنظيف الضرورية في وقت تجميع النفايات.

تأتي أهمية هذه الناحية من أن بعض المبرمجين، وبشكل خاص مبرمجي *C++*، قد يظنون خطأ أن الطريقة السابقة تشبه الطريقة *finalize()* المعرفة في *C++* والتي يتم استدعائها دائما عند الحاجة إلى تدمير عنصر.

طبعا الأمر مختلف تماما بين *C++* و *Java*، لأنه يجب القيام بتدمير العناصر في لغة *C++* دوماً، بينما قد لا يتم تدمير العناصر باستخدام مجمع النفايات في لغة جافا دوماً. لذلك عندما تحتاج للقيام بعملية ما أثناء عمالك (كعملية تدمير عنصر مثلاً)، يجب عليك القيام بها بنفسك. سأعطيك المثال التالي لتوضيح هذه الفكرة:

لنفترض أن عملية إنشاء عنصر تطلب أن يقوم برسم نفسه على الشاشة، فإذا لم تقم بمسح رسم العنصر من الشاشة فقد لا يتم ذلك أبداً لوحده، لأن مجمع النفايات قد يصبح كسولاً وينتهي البرنامج قبل أن يقوم حضرته بالتخلص من العناصر الشاردة.

لكن كيف يمكن إنجاز عملية المسح؟

من أجل مسح عنصر، يجب على المستخدم طلب طريقة المسح عند النقطة التي يكون بحاجة إليها.

وهذا الأمر جيد، لكنه قد يتعارض قليلا مع مفهوم التدمير *destruction* المتعارف عليه في *C++*. ففي هذه اللغة يجب القيام بتدمير جميع العناصر. أما إذا تم إنشاء عنصر بشكل محلي *local* في *C++*، أي تم إنشاؤه على مكس *stack* (هذا غير ممكن في جافا)، فإن عملية التدمير تتم أثناء إغلاق منطقة العمل التي تم فيها إنشاء هذا العنصر. أما إذا تم إنشاء العنصر في *C++* باستخدام *new* (كما في جافا)، فيتم طلب المدمر *Destructor* بطلب معامل التدمير *C++* وهو *Delete* (غير موجود في جافا). وإذا نسي المبرمج القيام بذلك فلن يتم طلب المدمر وسيبقى العنصر يسرح في الذاكرة مسببا ضعفها.

بينما لغة جافا لاتعطيك إمكانية إنشاء عناصر محلية، لأن عليك دوما استخدام *new*. كما أنها لاتستدعي طلب *delete* لتحرير العناصر لأن مجمع النفايات سيقوم بذلك عنك. لكن وجود مجمع النفايات لا يلغي الحاجة إلى استخدام المدمرات *Destructors*. ويجب ألا تستخدم الطريقة *finalize()* بشكل مباشر، وإنما يمكنك إنشاء طريقة تشبه مثلثتها في *C++*.

أحد الأشياء التي تفيدك فيها الطريقة *finalize()* هي مراقبة إجراء تجميع النفايات. يوضح المثال التالي ذلك، فتابع معنا:

```
//: Garbage.java
// Demonstration of the garbage
// collector and finalization
class Chair {
    static boolean gcrun = false;
    static boolean f = false;
    static int created = 0;
    static int finalized = 0;
    int i;
    Chair() {
        i = ++created;
        if(created == 47)
            System.out.println("Created 47");
    }
    protected void finalize() {
        if(!gcrun) {
```



```

gcrun = true;
System.out.println(
    "Beginning to finalize after " +
    created + " Chairs have been created");
}
if(i == 47) {
    System.out.println(
        "Finalizing Chair #47, " +
        "Setting flag to stop Chair creation");
    f = true;
}
finalized++;
if(finalized >= created)
    System.out.println(
        "All " + finalized + " finalized");
}
}
public class Garbage {
    public static void main(String[] args) {
        if(args.length == 0) {
            System.err.println("Usage: \n" +
                "java Garbage before\n or:\n" +
                "java Garbage after");
            return;
        }
        while(!Chair.f) {
            new Chair();
            new String("To take up space");
        }
        System.out.println(
            "After all Chairs have been created:\n" +
            "total created = " + Chair.created +
            ", total finalized = " + Chair.finalized);
        if(args[0].equals("before")) {
            System.out.println("gc()");
            System.gc();
            System.out.println("runFinalization()");
            System.runFinalization();
        }
    }
}

```

```

}
System.out.println("bye!");
if(args[0].equals("after"))
    System.runFinalizersOnExit(true);
}
} ///:~

```

يقوم المثال السابق بإنشاء العديد من عناصر *Chair*، وفي نقطة معينة بعد أن يبدأ مجمع النفائات بالعمل، يتوقف البرنامج عن إنشاء عناصر *Chair* جديدة. وعلى اعتبار أن مجمع النفائات قد يعمل في أي لحظة، لذلك لن نعرف تماما متى يبدأ بالعمل، لهذا السبب قمنا بتعريف العضو *gcrun* الذي يدلنا فيما إذا تم تشغيل مجمع النفائات أم لا. أما العضو الآخر *f*، فيستخدم لإخبار الحلقة الموجودة في *main()* بأنه يجب أن يتوقف عن إنشاء عناصر جديدة.

ويتم تحديد قيم العضوين السابقين ضمن *finalize()* الذي يتم طلبه أثناء عملية تجميع النفائات.

هنالك أيضا متحولان ساكنان *created* و *finalized* يستخدمان لتحديد عدد العناصر التي تم إنشاؤها أو تدميرها على الترتيب.

ولكل عنصر *Chair* متحول خاص *i int* لمعرفة رقم هذا العنصر، وذلك من أجل إيقاف عملية إنشاء عناصر جديدة في حال وصل عددها إلى ٤٧.

كل ما سبق يتم حدوثه في *main()* ضمن الحلقة:

```

while(!Chair.f) {
    new Chair();
    new String("To take up space");
}

```

لكنك قد ترغب بمعرفة كيفية إيقاف هذه الحلقة لأنه لا يوجد أي شيء يدل على تغيير قيمة *Chair.f* في الواقع فإن إجراء *finalize()* سيقوم بذلك عندما ينهي العنصر رقم ٤٧.

وعندما تقوم بتنفيذ البرنامج السابق، عليك تحديد أحد الوسيطين "before" أو "after". الوسيط "before" سيستدعي الطريقة *System.gc()* (لإجبار تنفيذ مجمع النفائات) ثم سيستدعي الطريقة *System.runFinalizatuion()*



من أجل تشغيل إجراءات الإنهاء. أما عند استخدام الوسيط "after" سيقوم البرنامج باستدعاء `System.runFinalizersOnExit()`.

تحديد القيم الابتدائية لعضو...

من الميزات الهامة التي أتت بها لغة جافا هي أنها تضمن لك تحديد القيمة الابتدائية لأي متحول قبل أن يتم استخدامه. ففي حالة تعريف المتحولات بشكل محلي ضمن طريقة، فإن جافا تضمن ذلك بإعطائها خطأ في وقت الترجمة `compile-time error`.
لذلك إذا كتبت :

```
void f() {
    int i;
    i++;
}
```

ستحصل على رسالة خطأ تقول لك بأن عليك تحديد قيمة ابتدائية للمتحول `i`. بالطبع كان باستطاعة المترجم تحديد قيمة افتراضية لهذا المتحول، إلا أنه من الأفضل ترك ذلك للمبرمج حتى يعرف مكان العلة في برنامجه ويحدد القيمة التي يرغب بإعطائها لهذا المتحول.

وفي حال كان أحد الأنماط الأولية `Primitive Types` عضو معطيات صف ما فإن الأمر سيختلف قليلاً، فباعتبار أنه يمكن لأي طريقة تحديد قيمة ابتدائية للمعطيات أو استخدام هذه المعطيات، فلن يكون من المفيد عملياً إجبار المستخدم على تحديد قيم ابتدائية لهذا النوع من المعطيات قبل استخدامها. لذلك فإن أي عضو معطيات أولي سيأخذ دائماً قيمة ابتدائية قبل استخدامه. يوضح البرنامج التالي القيم الابتدائية لأنماط المعطيات الأولية:

```
//: InitialValues.java
// Shows default initial values
class Measurement {
    boolean t;
    char c;
    byte b;
    short s;
```

سلسلة الرضا للمعلومات

```
int i;
long l;
float f;
double d;
void print() {
    System.out.println(
        "Data type Initial value\n" +
        "boolean " + t + "\n" +
        "char " + c + "\n" +
        "byte " + b + "\n" +
        "short " + s + "\n" +
        "int " + i + "\n" +
        "long " + l + "\n" +
        "float " + f + "\n" +
        "double " + d);
}
}
public class InitialValues {
    public static void main(String[] args) {
        Measurement d = new Measurement();
        d.print();
        /* In this case you could also say:
        new Measurement().print();
        */
    }
} ///:~
```

خرج البرنامج السابق سيكون على الشكل:

```
Data type Initial value
boolean false
char
byte 0
short 0
int 0
long 0
float 0.0
double 0.0
```



وستجد فيما بعد أنك عندما تقوم بتعريف مؤشر عنصر ضمن صف دون ربطه بشكل ابتدائي مع صف جديد، سيأخذ هذا المؤشر القيمة *null*.

كيف تقوم جافا بتحديد القيم الابتدائية للمتحولات الساكنة؟

تسمح لك لغة جافا بتجميع عمليات تحديد القيم الابتدائية للمتحولات الساكنة ضمن ما يسمى بكتل ساكنة *static block* ضمن صف، وهي تأخذ شكلا مشابها للمثال التالي:

```
class Spoon {
    static int i;
    static {
        i = 47;
    }
    // . . .
```

لذلك فهي تبدو كأنها طريقة، إلا أنها عبارة عن كلمة المفتاح *static* متبوعة بجسم الطريقة. ويتم تنفيذ هذا الترميز لمرة واحدة فقط عند إنشاء العنصر أو عند الوصول إلى عضو ساكن *static member* في الصف.

وماذا عن المتحولات غير الساكنة؟

يتم ذلك بشكل مشابه تماما للمتحولات الساكنة. لنأخذ المثال التالي:

```
//: Mugs.java
// Java 1.1 "Instance Initialization"
class Mug {
    Mug(int marker) {
        System.out.println("Mug(" + marker +
            ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}
public class Mugs {
    Mug cl;
```

```

Mug c2;
{
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2
        initialized");
}
Mugs () {
    System.out.println("Mugs()");
}
public static void main(String[] args) {
    System.out.println("Inside main()");
    Mugs x = new Mugs();
}
} ///:~

```

يمكنك ملاحظة أن عبارة تحديد القيم الابتدائية مشابهة تماما لتلك المتعلقة بالمتحولات الساكنة وهي:

```

{
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
}

```

تحديد القيم الابتدائية للمصفوفات...

كما نعلم فإن عملية تحديد القيم الابتدائية للمصفوفات في لغة C عملية مضجرة ومملة. أما في C++ فقد تم استخدام مايسمى بتجميع عمليات تحديد القيم الابتدائية لجعل العملية أكثر أمانا.

أما لغة جافا فلا تمتلك أي مجمع كما في C++ لأن أي شيء هنا عبارة عن عنصر. والمصفوفات في هذه اللغة هي ببساطة عبارة عن سلسلة من العناصر أو من الأنماط الأولية، وتمتلك جميعها نفس النمط ويتم حزمها سوياً ضمن اسم محدد وحيد. ومن أجل تعريف مصفوفة بسيطة يمكن كتابة:

```
int[] a1;
```



كما يمكن تعريفها على الشكل:

```
int a1[];
```

ولايسمح لك المترجم بتحديد حجم المصفوفة، وهذا يعيدنا إلى مفهوم المؤشر، فكل مالدريك حتى هذه النقطة هو مؤشر إلى مصفوفة ولا توجد أية مساحة محجوزة لهذه المصفوفة حتى الآن.

من أجل القيام بحجز مساحة لمصفوفة يجب عليك تحديد قيمتها الابتدائية كما في الشكل:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

لكن ما الفائدة من تعريف مؤشر مصفوفة قبل حجز هذه المصفوفة فعليا؟

في الواقع السبب هو أن باستطاعتك وضمن جافا ربط مصفوفة بأخرى على الشكل:

```
a2 = a1;
```

فما تقوم به فعليا هو نسخ مؤشر كما هو موضح في المثال:

```
//: Arrays.java
// Arrays of primitives.
public class Arrays {
    public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4, 5 };
        int[] a2;
        a2 = a1;
        for(int i = 0; i < a2.length; i++)
            a2[i]++;
        for(int i = 0; i < a1.length; i++)
            prt("a1[" + i + "] = " + a1[i]);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~
```

فكما ترى هنا أن *a1* قد أعطي قيمة ابتدائية أما *a2* فلا، وتم بعد ذلك ربط *a2* مع *a1*. الأهم من ذلك هو أن لغة جافا قد حلت مشكلة عويصة كانت تواجه مبرمجي *C* و *C++*، وهي مشكلة تجاوز حدود المصفوفة دون أن يعطي المترجم أية إشارة عن هذا التجاوز، تصل بعدها إلى مرحلة التنفيذ وتظهر لك المشاكل وقد تقوم بإلقاء حاسوبك من النافذة إذا لم تكن صبورا!!!

سلسلة الرضا للمعلومات

جافا كلغة عصرية تجاوزت هذه المشكلة ولن يسمح لك المترجم الخاص باللغة بتجاوز حدود المصفوفة.

لكن ماذا لو لم تكن تعرف عدد العناصر التي ستحتاجها في مصفوفتك أثناء قيامك بكتابة البرنامج؟

الأمر بسيط، قم فقط باستخدام *new* لإنشاء عناصر جديدة في مصفوفتك. ويمكنك القيام بذلك أيضا حتى لو كنت تتعامل مع مصفوفة من العناصر الأولية. انظر المثال التالي:

```
//: ArrayNew.java
// Creating arrays with new.
import java.util.*;
public class ArrayNew {
    static Random rand = new Random();
    static int pRand(int mod) {
        return Math.abs(rand.nextInt()) % mod;
    }
    public static void main(String[] args) {
        int[] a;
        a = new int[pRand(20)];
        prt("length of a = " + a.length);
        for(int i = 0; i < a.length; i++)
            prt("a[" + i + "] = " + a[i]);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~
```

وعلى اعتبار أن حجم المصفوفة قد تم تحديده بشكل عشوائي باستخدام الطريقة *pRand()*، لذلك فإن إنشاء المصفوفة سيتم في وقت التنفيذ. وسترى أيضا أن القيم الابتدائية للعناصر الأولية في المصفوفة ستأخذ القيم "empty" عند تنفيذ هذا البرنامج. يمكن بالطبع تعريف وتحديد المصفوفة في نفس التعليمة:

```
int[] a = new int[pRand(20)];
```

فإذا كنت تتعامل مع مصفوفة عناصر غير أولية، يتوجب عليك دوما استخدام *new*. انظر المثال:



```

//: ArrayClassObj.java
// Creating an array of non-primitive objects.
import java.util.*;
public class ArrayClassObj {
    static Random rand = new Random();
    static int pRand(int mod) {
        return Math.abs(rand.nextInt()) % mod;
    }
    public static void main(String[] args) {
        Integer[] a = new Integer[pRand(20)];
        prt("length of a = " + a.length);
        for(int i = 0; i < a.length; i++) {
            a[i] = new Integer(pRand(500));
            prt("a[" + i + "] = " + a[i]);
        }
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~

```

في البرنامج السابق، حتى لو تم طلب new لإنشاء المصفوفة:

```
Integer[] a = new Integer[pRand(20)];
```

فإنها ستكون مصفوفة مؤشرات فقط حتى يتم تحديد القيم الابتدائية لها بإنشاء عناصر جديدة

كما في الشكل:

```
a[i] = new Integer(pRand(500));
```

فإذا نسيت إنشاء عنصر، ستحصل على استثناء exception في وقت التنفيذ عندما

تحاول قراءة موقع المصفوفة الفارغة.

من الممكن أيضا تحديد القيم الابتدائية لمصفوفة على الشكل:

```

//: ArrayInit.java
// Array initialization
public class ArrayInit {
    public static void main(String[] args) {
        Integer[] a = {
            new Integer(1),
            new Integer(2),

```

```
new Integer(3),
};
// Java 1.1 only:
Integer[] b = new Integer[] {
    new Integer(1),
    new Integer(2),
    new Integer(3),
};
}
} ///:~
```

أما بالنسبة للمصفوفات متعددة الأبعاد فالأمر مشابه تماما لما رأيناه من قبل:

```
//: MultiDimArray.java
// Creating multidimensional arrays.
import java.util.*;
public class MultiDimArray {
    static Random rand = new Random();
    static int pRand(int mod) {
        return Math.abs(rand.nextInt()) % mod;
    }
    public static void main(String[] args) {
        int[][] a1 = {
            { 1, 2, 3, },
            { 4, 5, 6, },
        };
        for(int i = 0; i < a1.length; i++)
            for(int j = 0; j < a1[i].length; j++)
                prt("a1[" + i + "][" + j +
                    "]" = " + a1[i][j]);
        // 3-D array with fixed length:
        int[][][] a2 = new int[2][2][4];
        for(int i = 0; i < a2.length; i++)
            for(int j = 0; j < a2[i].length; j++)
                for(int k = 0; k < a2[i][j].length;
                    k++)
                    prt("a2[" + i + "][" +
                        j + "][" + k +
                        "]" = " + a2[i][j][k]);
    }
}
```



```
// 3-D array with varied-length vectors:
int[][][] a3 = new int[pRand(7)][][];
for(int i = 0; i < a3.length; i++) {
    a3[i] = new int[pRand(5)][];
    for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
}
for(int i = 0; i < a3.length; i++)
    for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length;
            k++)
            prt("a3[" + i + "][" +
                j + "][" + k +
                "] = " + a3[i][j][k]);
// Array of non-primitive objects:
Integer[][] a4 = {
    { new Integer(1), new Integer(2) },
    { new Integer(3), new Integer(4) },
    { new Integer(5), new Integer(6) },
};
for(int i = 0; i < a4.length; i++)
    for(int j = 0; j < a4[i].length; j++)
        prt("a4[" + i + "][" + j +
            "] = " + a4[i][j]);
Integer[][] a5;
a5 = new Integer[3][];
for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer[3];
    for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i*j);
}
for(int i = 0; i < a5.length; i++)
    for(int j = 0; j < a5[i].length; j++)
        prt("a5[" + i + "][" + j +
            "] = " + a5[i][j]);
}
static void prt(String s) {
    System.out.println(s);
}
```

}
} ///:~





المبرمج وبشكل دائم إلى التعامل مع المكتبات *Libraries*. لذلك يجب
 أن يكون قادراً على ربط الأجزاء التي يحتاجها. كذلك يجب عليه تحديد
 العناصر المتاحة لجميع الأشخاص أو التي سيتم إخفاؤها عن بعضهم.

يحتاج

التعامل مع الحزم Packages...

الحزمة هي ماتحصل عليه عندما تستخدم تعليمة `import` لجلب كامل مكتبة ما، مثلاً:

```
import java.util.*;
```

وعندما تحتاج إلى جلب صف وحيد، فبإمكانك تسمية الصف المطلوب على الشكل:

```
import java.util.Vector;
```

وعندما تقوم بإنشاء ملف ترميز بلغة جافا، وهو ما نسميه بوحدة ترجمة `compilation unit`، يجب أن تمتلك كل من وحدات الترجمة هذه اسماً ينتهي بـ `.java`. ويمكن أن تتضمن صفوفاً عامة لها نفس الاسم.

وبعد أن تقوم بترجمة ملف `java` ستحصل على ملف خرج بنفس الاسم لكن بالامتداد `class` لكل صف من الصفوف الموجودة في الملف. أي أن برنامج جافا عبارة عن حزمة من ملفات `class` التي يمكن تجميعها وضغطها في ملف `JAR` باستخدام الأداة الموافقة ضمن جافا.

كذلك فإن المكتبة `library` عبارة عن مجموعة من ملفات الصفوف هذه. وبتلك كل ملف صفافاً واحداً عاماً `public`. ويمكنك تجميع هذه الملفات في حزمة واحدة، فعندما نكتب:

```
package mypackage;
```

ذلك في بداية الملف فأنت تقوم بوضع وحدة الترجمة هذه على شكل مكتبة بالاسم `mypackage`، ويستطيع بعد ذلك أي كان استخدام هذه المكتبة بتضمينها ضمن برنامج بكتابة:

```
import mypackage;
```

لنفترض مثلاً أن لدينا ملفاً بالاسم `MyClass.java`. هذا يعني بأن هناك صفافاً عاماً `public` واحداً واحداً فقط، ويجب أن يكون اسم هذا الصف `MyClass` وذلك على الشكل:

```
package mypackage;
public class MyClass {
// . . .
```


الآن عندما يرغب أيّ كان باستخدام الصف *MyClass*، يجب عليه استخدام كلمة المفتاح *import* لجعل الصفوف الموجودة في الحزمة *mypackage* متاحة للآخرين. فيمكن مثلاً كتابة العبارة:

```
mypackage.MyClass m = new mypackage.MyClass();
ويمكن لكلمة المفتاح import أن تجعل ذلك أوضح:
```

```
import mypackage.*;
// . . .
```

```
MyClass m = new MyClass();
```

ويمكن وضع جميع ملفات *.class* التي تولّف حزمة *package* ضمن مجلد نظام واحد لسهولة الوصول إليها.

ولحلّ مشكلة إنشاء اسم حزمة وحيد، يقوم النظام بتضمين اسم مجال إنترنت معكوساً للحساب الذي قام بإنشاء الصف وذلك قبل اسم الحزمة. أما لتحويل اسم هذه الحزمة إلى طريق *path* يستطيع المفسّر *Interpreter* من خلاله إيجاد صفوف الحزمة فهو يتبع الطريقة التالية:

يقو أولاً بالبحث عن متحول البيئة *CLASSPATH* الذي يحتوي على مجلد أو أكثر يعتبر كجذر *Root* للبحث عن ملفات *.class*. وابتداءً من هذا الجذر يأخذ المفسّر اسم الحزمة ويستبدل كل نقطة فيه بـ *./*. فمثلاً الحزمة *foo.bar.baz* ستصبح *foo/bar/baz* أو *foo\bar\baz* حسب نظام التشغيل. ثم يدمج الاسم الناتج مع المداخل المختلفة الموجودة في *CLASSPATH*.

لتوضيح ذلك بشكل أفضل لنفترض أن اسم المجال *noukari.com*، الآن وبعد قلبه سيصبح *com.noukari*، ومن أجل إنشاء مكتبة بالاسم *util* نكتب:

```
package com.noukari.util;
```

يمكن استخدام هذه الحزمة لإنشاء الملفين، الأول:

```
//: Vector.java
// Creating a package
package com.noukari.util;
public class Vector {
    public Vector() {
        System.out.println(
```

```
"com.noukari.util.Vector");
}
} ///:~
```

أما الملف الثاني:

```
//: List.java
// Creating a package
package com.noukari.util;
public class List {
    public List() {
        System.out.println(
            "com.noukari.util.List");
    }
} ///:~
```

وسيتّم وضع الملفين السابقين في المجلّد الفرعي:

C:\DOC\JavaT\com\noukari\util

وذلك بافتراض أن:

CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT

محدّدات الوصول Access

...Specifiers

كما ذكرنا من قبل، فإنّ محدّدات الوصول *access specifiers* في جافا هي *public* و *protected* و *private*. ويتم وضع هذه المحدّدات قبل تعريف أيّ عضو من الصف سواء أكان عضو معطيات *data member* أو طريقة *method*.

كلّ محدّد من هذه المحدّدات يتحكّم فقط بعملية الوصول إلى العضو المعرّف، وهو ما يختلف عن C++ حيث يبقى هذا المحدّد معرّفاً لجميع الأجزاء حتى ظهور محدّد آخر.



المحدد الصديق *...Friendly*

في جميع الأمثلة السابقة لم نقم بوضع أيّ محدّد، في هذه الحالة كأننا نقول بأنّ العضو المحدّد يمتلك نمط الوصول *Friendly*. هذا النمط يعني بأنّه يمكن لجميع الصفوف الأخرى في الحزمة الوصول إلى العضو الصديق *friendly member*، أما الصفوف الموجودة خارج الحزمة فلن تستطيع ذلك.

المحدد العام *...Public*

هذا النمط يعني بأنّه يمكن لجميع الصفوف الأخرى في الحزمة وخارجها الوصول إلى العضو العام *public member*. أضف فقط كلمة المفتاح *public* قبل تعريف هذا العضو. لنفترض مثلاً أنّ لديك الحزمة *dessert* المعرفة على الشكل:

```
//: Cookie.java
// Creates a library
package c05.dessert;
public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    void foo() { System.out.println("foo"); }
} ///:~
```

تستطيع الآن استخدام الحزمة السابقة في البرنامج التالي:

```
//: Dinner.java
// Uses the library
import c05.dessert.*;
public class Dinner {
    public Dinner() {
        System.out.println("Dinner constructor");
    }
    public static void main(String[] args) {
```

```

Cookie x = new Cookie();
//! x.foo(); // Can't access
}
} ///:~

```

وتستطيع هنا إنشاء عنصر *Cookie*، لأنّ باني هذا العنصر من نمط عام *public* كما أنّ الصف هو أيضاً من نمط عام *public*. أما العضو *foo()* فلن تستطيع الوصول إليه ضمن البرنامج *Dinner.java* لأنّ هذا العضو صديق فقط ضمن الحزمة *dessert*.

سأعطيك الآن مثلاً عجيبيّاً!!

سنأخذ المثال التالي الذي سيبدو للوهلة الأولى بأنّه قد كسر القواعد السابقة، لنفترض أنّ لدينا الملف التالي في مجلد معين:

```

//: Cake.java
// Accesses a class in a separate
// compilation unit.
class Cake {
    public static void main(String[] args) {
        Pie x = new Pie();
        x.f();
    }
} ///:~

```

وكذلك لدينا ملفاً آخر ضمن نفس المجلد:

```

//: Pie.java
// The other class
class Pie {
    void f() { System.out.println("Pie.f()"); }
} ///:~

```

سيبدو لك الملفان السابقان غريبين، وقد تستغرب كيف يستطيع *Cake* إنشاء عنصر *Pie* واستدعاء الطريقة *f()*.



لكن كل ماسبق ذكره صحيح لأن الملفين السابقين موجودان في نفس المجلد، لذلك تعاملهما جافا وكأنهما جزء من حزمة افتراضية `default package` لهذا المجلد، وسيكونان لهذا السبب صديقين.

النمط الخاص **Private**...

يمكن استخدام كلمة المفتاح `private` لجعل عضو ما خاصاً فقط بهذا الصف، أما بقية الصفوف في الحزمة فلا يمكنها الوصول إليه. كمثال على ذلك:

```
//: IceCream.java
// Demonstrates "private" keyword
class Sundae {
    private Sundae() {}
    static Sundae makeASundae() {
        return new Sundae();
    }
}
public class IceCream {
    public static void main(String[] args) {
        //! Sundae x = new Sundae();
        Sundae x = Sundae.makeASundae();
    }
} //:~
```

تلاحظ في المثال السابق أنه لا يمكنك إنشاء عنصر `Sundae`، بينما تستطيع بدلاً من ذلك استدعاء الطريقة `makeASundae()` لإنشاء هذا العنصر.

النمط المحمي **Protected**...

لنفترض أنك قمت بإنشاء حزمة جديدة، ثم قمت بتوريث صف موجود في حزمة أخرى، في هذه الحالة يمكنك الوصول إلى الأعضاء ذوي النمط `public` في الحزمة الأساسية. لكن هنالك حالات نحتاج فيها إلى منح سماحية الوصول إلى عضو خاص من قبل الصفوف

المشتقة *derived classes* فقط. عندها نعطي هذا العضو السماحية *protected*، كمثال على ذلك:

```
//: ChocolateChip.java
// Can't access friendly member
// in another class
import c05.dessert.*;
public class ChocolateChip extends Cookie {
    public ChocolateChip() {
        System.out.println(
            "ChocolateChip constructor");
    }
    public static void main(String[] args) {
        ChocolateChip x = new ChocolateChip();
        /// x.foo(); // Can't access foo
    }
} ///:~
```

حيث تفيد *extends* في التوريث من صف أساسي معرف من قبل. ومن أجل إتاحة الوصول إلى الطريقة *foo()*، يجب تغيير الصف *Cookie* على الشكل:

```
public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    protected void foo() {
        System.out.println("foo");
    }
}
```

وغالباً ما يرتبط مفهوم التحكم بالوصول إلى العناصر بما نسميه إخفاء التنفيذ *implementation hiding* أو الكبسلة *Encapsulation*.



لكن ماذا بالنسبة إلى تحديد سماحية الوصول إلى الصفوف؟

يمكن لكل صف أن يكون عاماً `public` أو صديقاً `Friendly`، ولا يمكن أبداً أن يكون خاصاً `private` (مما يجعله غير متاح إلا لنفسه) أو حتى محمياً `protected`. أما إذا كنت ترغب بالوصول إلى صف آخر بالوصول إلى صف فاجعل جميع بيانات هذا الصف خاصة `private`، مما يمنع أيّاً كان (عداك) من إنشاء عنصر ضمن الصف. المثال التالي يوضح ماسبق ذكره:

```
//: Lunch.java
// Demonstrates class access specifiers.
// Make a class effectively private
// with private constructors:
class Soup {
    private Soup() {}
    // (1) Allow creation via static method:
    public static Soup makeSoup() {
        return new Soup();
    }
    // (2) Create a static object and
    // return a reference upon request.
    // (The "Singleton" pattern):
    private static Soup ps1 = new Soup();
    public static Soup access() {
        return ps1;
    }
    public void f() {}
}
class Sandwich { // Uses Lunch
    void f() { new Lunch(); }
}
// Only one public class allowed per file:
public class Lunch {
    void test() {
        // Can't do this! Private constructor:
        //! Soup priv1 = new Soup();
        Soup priv2 = Soup.makeSoup();
    }
}
```

```

Sandwich f1 = new Sandwich();
Soup.access().f();
}
} ///:~

```

الصفوف ...Classes

لقد كانت المشكلة الأساسية في لغات البرمجة الإجرائية، كـ *C* مثلاً، هي أننا وعند الحاجة إلى استخدام إجرائية مكتوبة سابقاً، مع إجراء تغيير قد يكون بسيطاً جداً، كنا نقوم بإعادة نسخ الإجرائية وإجراء التغيير المطلوب وحفظ هذه الإجرائية باسم جديد. لذلك من أهم القضايا التي تمحورت حولها البرمجة غرضية التوجه بشكل عام، وجافاً بشكل خاص، كانت قضية إعادة استخدام الترميز *code reuse* والتي تتم بإنشاء صفوف جديدة بالاعتماد على صفوف موجودة مسبقاً وتم اختبارها. توجد طريقتان للقيام بذلك:

- ✓ الأولى تعتمد على إنشاء عناصر لصف موجود مسبقاً في الصف الجديد، وتسمى هذه الطريقة بالتركيب *Composition* لأن الصف الجديد مركب من عناصر الصفوف الموجودة مسبقاً.
- ✓ أما الثانية فتعتمد على إنشاء الصف الجديد كنمط *type* لصف موجود مسبقاً، حيث تقوم هنا بأخذ نموذج الصف الموجود وإضافة ترميز إليه دون التعديل على هذا الصف. هذه الطريقة تدعى بالتوريث *Inheritance*.

التركيب ...Composition

وكما ذكرنا مسبقاً فأنت تقوم هنا بوضع مؤشرات العناصر في صفوف جديدة. لنفترض مثلاً أنك ترغب بالتعامل مع عدة عناصر *String*، اثنان أوليان *primitive* أما الثالث فهو عنصر من صف آخر. بالنسبة للعنصر غير الأولي قم فقط بوضع مؤشر إليه في الصف الجديد، أما بالنسبة للصفوف الأولية فقم بتعريفها في صفك فقط:




```

//: SprinklerSystem.java
// Composition for code reuse
package c06;
class WaterSource {
    private String s;
    WaterSource() {
        System.out.println("WaterSource()");
        s = new String("Constructed");
    }
    public String toString() { return s; }
}
public class SprinklerSystem {
    private String valve1, valve2, valve3,
        valve4;
    WaterSource source;
    int i;
    float f;
    void print() {
        System.out.println("valve1 = " + valve1);
        System.out.println("valve2 = " + valve2);
        System.out.println("valve3 = " + valve3);
        System.out.println("valve4 = " + valve4);
        System.out.println("i = " + i);
        System.out.println("f = " + f);
        System.out.println("source = " + source);
    }
    public static void main(String[] args) {
        SprinklerSystem x = new SprinklerSystem();
        x.print();
    }
} ///:~

```

تلاحظ ضمن الصف `WaterSource` وجود طريقة خاصة هي `toString()`. ستتعلم لاحقاً أن لكل عنصر غير أولي طريقة `toString()` يتم استدعاؤها في حالات خاصة عندما يتوقع المترجم عنصراً من نمط `String` فيفاجأ بعنصر من نمط آخر. فمثلاً في التعبير التالي:

```
System.out.println("source = " + source);
```

يرى المترجم أنك تحاول دمج عنصر من نمط *String* ("source=") إلى عنصر من نمط *WaterSource*. لذلك يقول لنفسه "سأقوم بتحويل *source* إلى *String* باستدعاء *toString()*"، بعد ذلك يستطيع دمج عنصر *String* وتمرير النتيجة إلى *System.out.println()*.

أست معي بأن مترجم جافا أكثر ذكاءً مما نتصور!!
ستجد كذلك بأن المترجم سيقوم تلقائياً بإنشاء عناصر لكل من المؤشرات المعرّقة في البرنامج السابق، فمثلاً عند استدعاء الباني الافتراضي للصف *WaterSource* من أجل توليد العنصر *source*، سيظهر خرج تعليمة الطباعة على الشكل:

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null
```

أي أن الحقول الأولية في الصف ستأخذ 0 كقيمة ابتدائية، أما مؤشرات العناصر فتأخذ القيمة *Null*.

يمكنك تحديد القيمة الابتدائية للمؤشرات في أحد الأماكن التالية:

١. أثناء تعريف العناصر، أي أن القيمة الابتدائية ستأخذ دوماً قبل استدعاء الباني.
 ٢. ضمن باني الصف.
 ٣. قبل استخدام العنصر مباشرة، مما يقلل الوقت الإضافي المطلوب خاصة في الحالات التي لانتاج فيها إلى إنشاء العنصر.
- يوضح المثال التالي كيفية استخدام الحالات التالية:

```
//: Bath.java
// Constructor initialization with composition
class Soap {
    private String s;
    Soap() {
        System.out.println("Soap() ");
        s = new String("Constructed");
    }
}
```

```

    }
    public String toString() { return s; }
}
public class Bath {
    private String
    // Initializing at point of definition:
    s1 = new String("Happy"),
    s2 = "Happy",
    s3, s4;
    Soap castille;
    int i;
    float toy;
    Bath() {
        System.out.println("Inside Bath() ");
        s3 = new String("Joy");
        i = 47;
        toy = 3.14f;
        castille = new Soap();
    }
    void print() {
        // Delayed initialization:
        if(s4 == null)
            s4 = new String("Joy");
        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
        System.out.println("s3 = " + s3);
        System.out.println("s4 = " + s4);
        System.out.println("i = " + i);
        System.out.println("toy = " + toy);
        System.out.println("castille = " +
            castille);
    }
    public static void main(String[] args) {
        Bath b = new Bath();
        b.print();
    }
} //:~

```

لاحظ هنا أن تعليمة `Bath` تنفذ قبل أن يتم تحديد أي قيمة ابتدائية. أما خرج البرنامج فسيكون على الشكل:

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

التوريث ...Inheritance

تعتبر مسألة التوريث من القضايا الأساسية في البرمجة غرضية التوجه، وعندما نقوم بالتوريث، فكأنك تقول "هذا الصف الجديد يشبه ذاك الصف القديم". وبذلك نقوم ببساطة بإعطاء اسم الصف كالعادة، لكنك قبل فتح قوس بداية جسم الصف، تضع كلمة المفتاح `extends` تتبعها باسم الصف الأساسي. يؤدي ذلك وبشكل تلقائي إلى جلب جميع أعضاء المعطيات وجميع الطرق الموجودة في الصف الأساسي. لنأخذ المثال التالي:

```
//: Detergent.java
// Inheritance syntax & properties
class Cleanser {
    private String s = new String("Cleanser");
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public void print() { System.out.println(s); }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        x.print();
    }
}
```



```

public class Detergent extends Cleanser {
// Change a method:
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub(); // Call base-class version
    }
// Add methods to the interface:
    public void foam() { append(" foam()"); }
// Test the new class:
    public static void main(String[] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        x.print();
        System.out.println("Testing base class:");
        Cleanser.main(args);
    }
} ///:~

```

من المثال السابق يمكنك ملاحظة عدة أمور:

في الطريقة `Cleaner.append()` استخدم المعامل `+=` لدمج عناصر `String` إلى `s`. وهي إحدى الأمور التي قامت جافا بتوسيعها للعمل مع السلاسل `Strings`. لاحظ أيضاً استخدام الطريقة `main()` في الصفتين `Cleanser` و `Detergent`. لذلك يمكن إنشاء `main()` في كل صف من صفوفك، وهو مائنصح به غالباً لمساعدتك في اختبار ترميز الصفوف. لكن حتى لو كان لديك الكثير من الصفوف في برنامجك فسيتم طلب الطريقة `main()` فقط للصف العام `public` الذي يتم استدعاؤه في سطر الأوامر.

فمثلاً إذا طلبنا `java Detergent`، فسيتم استدعاء `Detergent.main()`. أما إذا طلبنا `java Cleanser` فسيتم استدعاء `Cleanser.main()` على الرغم من أن `Cleanser` ليس صفّاً عاماً.

هذه التقنية المستخدمة والتي تتطلب وضع `main()` في كل صف، تسمح لك باختبار كل صف بشكل سهل. وحتى لو تم الانتهاء من الاختبار فليست بحاجة إلى حذف `main()` لأنك قد تحتاج إليها في اختبارات قادمة. لاحظ أيضاً أن `Detergent.main()` يستدعي `Cleanser.main()` بشكل صريح.

كذلك فإنه من الضروري جعل جميع طرق `Cleanser` بحالة عامة `public`. وتذكر بأنك إذا لم تضع أيّ محدّد وصول `access specifier` إلى العضو فسيأخذ المحدّد الافتراضي `Friendly` والذي يسمح فقط لأعضاء الحزمة بالوصول إليه. لذلك وضمن هذه الحزمة، يمكن لأيّ كان استخدام هذه الطرق عندما لا يتم وضع محدّد الوصول، بالتالي لن يعاني `Detergent` من أية مشاكل. لكن عندما يحتاج أيّ صف من حزمة أخرى إلى الوراثة من `Cleanser` فلن يصل إلا إلى الأعضاء العاميين `public` فقط.

ولذلك للتخطيط من أجل التوريث، ومبدأ عام ضع جميع الحقول محمية `protected` أما الطرق فاتركها عامة `public`.

لاحظ أيضاً بأن الصف `Cleanser` يمتلك مجموعة من الطرق في واجهته وهي : `append()`, `dilute()`, `apply()`, `scrub()`, `print()` وعلى اعتبار أن الصف `Detergent` هو صف مشتق `Derived` من الصف `Cleanser`، فإنه سيحصل تلقائياً على جميع هذه الطرق، حتى لو لم ترها معرفة بشكل صريح في الصف `Detergent`.

كذلك فإنه من الممكن أخذ طريقة تمّ تعريفها في الصف الأساسي وإجراء التعديلات عليها في الصف المشتق كما ترى في الطريقة `scrub()`. وعندما تحتاج ضمن الصف المشتق إلى طلب الطريقة المعرفة أصلاً ضمن الصف الأساسي، قم فقط بإضافة كلمة المفتاح `super` قبل اسم الطريقة. فمثلاً التعبير `super.scrub()` سيقوم باستدعاء الطريقة `scrub()` المعرفة في الصف الأساسي `Cleanser` وليست تلك المعرفة في الصف المشتق `Detergent`.



تستطيع أيضاً في أي صف مشتق إضافة طرق جديدة غير موجودة أصلاً في الصف الأساسي. لذلك تستطيع في `Detergent.main()` استدعاء جميع الطرق المتاحة في الصف `Cleanser` إضافة إلى تلك المعرفة طبعاً في الصف `Detergent` (كالطريقة `foam()` مثلاً).

طبعاً وبشكل منطقي يتم استدعاء بانينات الصف الأساسي أولاً، ثم بانينات الصفوف المشتقة. يوضح المثال التالي ذلك:

```
//: Cartoon.java
// Constructor calls during inheritance
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}
class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing
        constructor");
    }
}
public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon
        constructor");
    }
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
} ///:~
```

وسيطهر خرج البرنامج السابق على الشكل:

```
Art constructor
Drawing constructor
Cartoon constructor
```

يوضح المثال السابق كيفية استدعاء بانيات الصفوف بدون وسطاء، والتي يمكن للمترجم استدعاءها بسهولة. أما في حال أردت استدعاء باني صف أساسي يمتلك وسطاء، يجب عندها، وبشكل صريح، كتابة الطلب باستخدام *super* مع قائمة الوسطاء المحددة، مثلاً:

```
//: Chess.java
// Inheritance, constructors and arguments
class Game {
    Game(int i) {
        System.out.println("Game constructor");
    }
}
class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        System.out.println("BoardGame constructor");
    }
}
public class Chess extends BoardGame {
    Chess() {
        super(11);
        System.out.println("Chess constructor");
    }
    public static void main(String[] args) {
        Chess x = new Chess();
    }
} ///:~
```

وإذا لم تقم باستدعاء باني الصف الأساسي في *BoardGame()*، سيعترض المترجم على عدم إيجاده باني النموذج *Game()*. إضافة إلى ذلك يجب أن تكون عملية استدعاء باني الصف الأساسي ضمن باني الصف المشتق أوّل عمل تقوم به.



أصبح أخيراً هناك معنى لاستخدام محدد الوصول

...protected

كما ذكرنا سابقاً، فإن نمط الوصول *protected* يستخدم عند التوريث *Inheritance* للسماح لأعضاء صف بأن تكون مخفية عن العالم الخارجي، بينما تكون ظاهرة للصفوف المشتقة.

يوضح المثال التالي أهمية استخدام هذا النمط:

```
//: Orc.java
// The protected keyword
import java.util.*;
class Villain {
    private int i;
    protected int read() { return i;}
    protected void set(int ii) { i = ii; }
    public Villain(int ii) { i = ii; }
    public int value(int m) { return m*i; }
}
public class Orc extends Villain {
    private int j;
    public Orc(int jj) { super(jj); j = jj; }
    public void change(int x) { set(x); }
} ///:~
```

أصبح أيضاً هناك معنىً للتحميل للأعلى *Upcasting*...

إن أهمية التوريث لا تقتصر على تزويد الصفوف الجديدة بالطرق المعروفة مسبقاً فقط، وإنما أيضاً في نوعية العلاقة التي يمكن التعبير عنها بين الصف الجديد والصف الأساسي. تتلخص هذه العلاقة بالقول "الصف الجديد عبارة عن نمط *of type* من الصف المعروف مسبقاً".

والأمر الهام أيضاً والذي تم ذكره مسبقاً هو تحميل الطرق للأعلى *Upcasting* والذي يمكننا من الانتقال من النمط الأكثر خصوصية إلى النمط الأكثر عمومية.

يوضح المثال التالي ماسبق ذكره:

```
//: Wind.java
// Inheritance & upcasting
import java.util.*;
class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}
// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Upcasting
    }
} ///:~
```

لاحظ الشيء الهام في المثال السابق وهو استخدام مؤشر إلى *Instrument* كوسيط للطريقة *tune()*.



Final أخيراً وليس آخراً...

عندما نستخدم كلمة المفتاح *Final* فنعني بها أن "ذلك الشيء لا يمكن تغييره".
توضح الفقرات التالية الأماكن الثلاثة التي نستخدم فيها *Final*.

المعطيات النهائية *Final Data*...

وهي مانسميها بالثوابت *Constant*. فعندما نستخدم *Final* مع المعطيات الأولية *primitive* فإنها تجعل القيمة ثابتة، أما عندما نستخدمها مع مؤشر عنصر *Object handle* فتجعل المؤشر ثابتاً.

يوضح المثال التالي كيفية استخدام هذا النوع من الحقول:

```
//: FinalData.java
// The effect of final on fields
class Value {
    int i = 1;
}
public class FinalData {
    // Can be compile-time constants
    final int i1 = 9;
    static final int I2 = 99;
    // Typical public constant:
    public static final int I3 = 39;
    // Cannot be compile-time constants:
    final int i4 = (int) (Math.random() * 20);
    static final int i5 =
        (int) (Math.random() * 20);
    Value v1 = new Value();
    final Value v2 = new Value();
    static final Value v3 = new Value();
    //! final Value v4; // Pre-Java 1.1 Error:
    // no initializer
```

```
// Arrays:
final int[] a = { 1, 2, 3, 4, 5, 6 };
public void print(String id) {
    System.out.println(
        id + ": " + "i4 = " + i4 +
        ", i5 = " + i5);
}
public static void main(String[] args) {
    FinalData fd1 = new FinalData();
    ///! fd1.i1++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
    fd1.v1 = new Value(); // OK -- not final
    for(int i = 0; i < fd1.a.length; i++)
        fd1.a[i]++; // Object isn't constant!
    ///! fd1.v2 = new Value(); // Error: Can't
    ///! fd1.v3 = new Value(); // change handle
    ///! fd1.a = new int[3];
    fd1.print("fd1");
    System.out.println("Creating new
    FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");
}
} ///:~
```

ويعطي هذا البرنامج الخرج التالي:

```
fd1: i4 = 15, i5 = 9
Creating new FinalData
fd1: i4 = 15, i5 = 9
fd2: i4 = 10, i5 = 9
```



الطرق النهائية *Final Methods*...

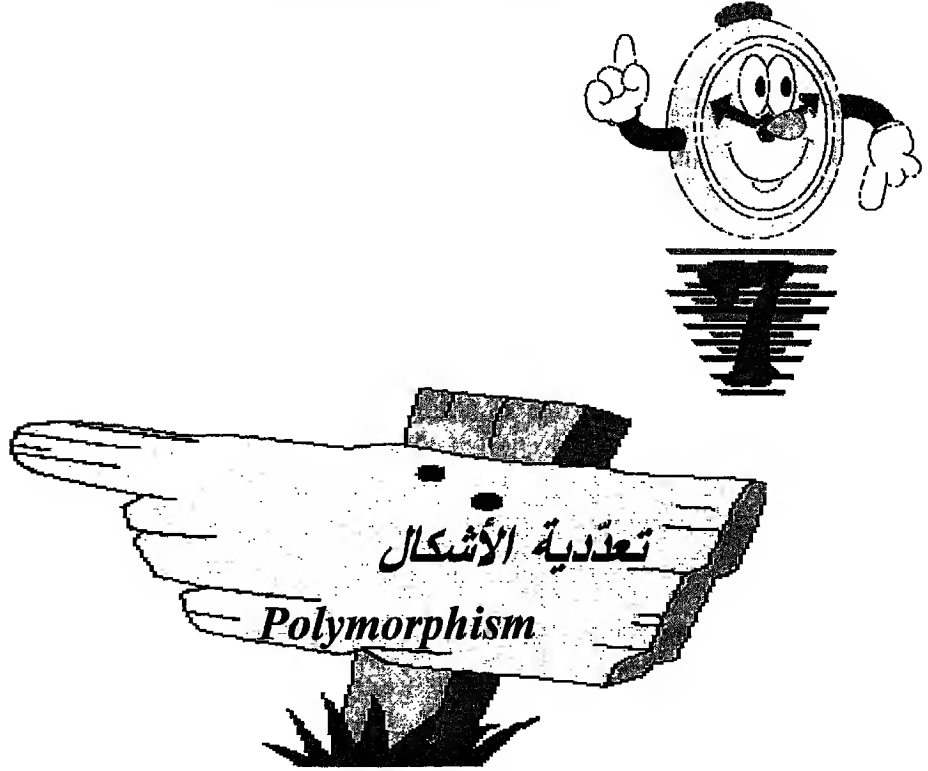
يوجد سببان لاستخدام هذا النوع من الطرق، الأول وضع قفل "lock" على الطريقة لمنع أي صف مورث من تغيير معنى الطريقة. أما السبب الثاني فهو لجعل الطرق أكثر فعالية لتحويل أي استدعاء للطريقة إلى استدعاء داخلي *inline*، مما يلغي الطريقة الاعتيادية التي يستخدمها المترجم لمعالجة هذا الطلب (دفع الوسطاء إلى المكثس، ثم جلب جسم الطريقة وتنفيذه، ثم مسح الوسطاء من المكثس وأخيراً معالجة قيمة الإرجاع)، ويستخدم بدلاً منها طريقة نسخ الترميز الحالي في جسم الطريقة.

الصفوف النهائية *Final Classes*...

عندما نقول بأن صفّاً ما هو صف نهائي *Final Class*، فهذا يعني بأنك لا ترغب بأن ترث من هذا الصف أو لا تريد السماح لأي كان بالقيام بذلك. كمثال على ذلك:

```
//: Jurassic.java
// Making an entire class final
class SmallBrain {}
final class Dinosaur {
    int i = 7;
    int j = 1;
    SmallBrain x = new SmallBrain();
    void f() {}
}
//! class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'
public class Jurassic {
    public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
} ///:~
```

لاحظ بأن المعطيات الأعضاء *Data Members* يمكن أن تكون نهائية *Final* أو غير نهائية.



مِيزة تعددية الأشكال *Polymorphism* من الأمور الأساسية التي تميّز
تعتبر لغات البرمجة غرضية التوجّه بعد التجريد *abstraction* والتوريث
Inheritance.
 وتسمح تعددية الأشكال بالتمييز بين نمط ونمط آخر مشابه له، على الرغم من أنهما مشتقان
 من نمط أساسي واحد.

التوجيه للأعلى ...Upcasting

كما رأينا في الفصول السابقة فإن التوجيه للأعلى *Upcasting* هو معالجة مؤشر عنصر *Object Handle* كمؤشر للنمط الأساسي. وسمي بالتوجيه للأعلى لأن عملية التوريث في الشجرة تتم نحو الصف الأساسي في القمة. يوضح المثال التالي كيفية استخدام التوجيه للأعلى:

```
//: Music.java
// Inheritance & upcasting
package c07;
class Note {
    private int value;
    private Note(int val) { value = val; }
    public static final Note
    middleC = new Note(0),
    cSharp = new Note(1),
    cFlat = new Note(2);
} // Etc.
class Instrument {
    public void play(Note n) {
        System.out.println("Instrument.play()");
    }
}
// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
    // Redefine interface method:
    public void play(Note n) {
        System.out.println("Wind.play()");
    }
}
public class Music {
    public static void tune(Instrument i) {
        // ...
        i.play(Note.middleC);
    }
}
```




```

public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Upcasting
}
} ///:~

```

لاحظ في هذا المثال أن الطريقة `Music.tune()` تقبل مؤشر نحو `Instrument`، كذلك نحو أي عنصر مشتق من `Instrument`، وهو ما يمكن ملاحظته في الطريقة `main()` عندما قمنا بتمرير مؤشر `Wind` للطريقة `tune()`.

لكن ما الفائدة من التوجيه للأعلى؟

قد يبدو لك المثال السابق غريباً بعض الشيء والسبب في ذلك هو أن عليك نسيان نمط العنصر. وقد نقول أليس من الأفضل لو أن الطريقة `tune()` تأخذ مؤشر `Wind` مباشرة كوسيط لها، لكن هذا سيؤدنا إلى نقطة أساسية: لو أننا قمنا بذلك فسنحتاج إلى كتابة طريقة `tune()` جديدة لكل نمط `Instrument` في النظام. لنفترض مثلاً أننا اتبعنا هذه الفكرة الجديدة وقمنا بإضافة الآلات `Stringed` و `Brass` كما في المثال التالي:

```

//: Music2.java
// Overloading instead of upcasting
class Note2 {
    private int value;
    private Note2(int val) { value = val; }
    public static final Note2
        middleC = new Note2(0),
        cSharp = new Note2(1),
        cFlat = new Note2(2);
} // Etc.
class Instrument2 {
    public void play(Note2 n) {
        System.out.println("Instrument2.play()");
    }
}
class Wind2 extends Instrument2 {

```

```

    public void play(Note2 n) {
        System.out.println("Wind2.play()");
    }
}
class Stringed2 extends Instrument2 {
    public void play(Note2 n) {
        System.out.println("Stringed2.play()");
    }
}
class Brass2 extends Instrument2 {
    public void play(Note2 n) {
        System.out.println("Brass2.play()");
    }
}
public class Music2 {
    public static void tune(Wind2 i) {
        i.play(Note2.middleC);
    }
    public static void tune(Stringed2 i) {
        i.play(Note2.middleC);
    }
    public static void tune(Brass2 i) {
        i.play(Note2.middleC);
    }
    public static void main(String[] args) {
        Wind2 flute = new Wind2();
        Stringed2 violin = new Stringed2();
        Brass2 frenchHorn = new Brass2();
        tune(flute); // No upcasting
        tune(violin);
        tune(frenchHorn);
    }
} //:~

```

البرنامج السابق سيعمل وستحصل على النتائج المرغوبة، لكن هناك نقطة سلبية كبيرة فيه هي أن عليك كتابة طريقة خاصة لكل صف *Instrument2* جديد تقوم بإنشائه. وهذا يعني المزيد من أسطر الترميز أولاً، والأهم من ذلك هو أنك عندما ترغب بإضافة طريقة



جديدة مثل (*tune*) أو نمط جديد من *Instrument* سيتوجب عليك القيام بعمل كبير.

ألا ترى معي يا صديقي من الأفضل القيام بكتابة طريقة واحدة تأخذ الصف الأساسي كوسيط لها بدلاً من الصفوف المشتقة؟ هذا بالضبط ما تسمح به خاصية تعددية الأشكال *Polymorphism*.

سنعود الآن إلى البرنامج الأول *Music.java* وسنحاول تنفيذه، سنحصل في النتيجة على *Wind.play()* وهو طبعاً الخرج المطلوب، لكن لننظر أكثر إلى الطريقة:

```
public static void tune(Instrument i) {
    // ...
    i.play(Note.middleC);
}
```

سيصل مؤشر *Instrument* إلى هذه الطريقة. لكن كيف سيتمكن المترجم من معرفة أن مؤشر *Instrument* هذا يدل على *Wind* هنا وليس *Brass* أو *Stringed*. الجواب هنا هو أن المترجم لا يستطيع معرفة ذلك، والحل هو باستخدام ما يسمى بالربط *Binding*.

الربط *Binding*؟

وهي عملية ربط استدعاء طريقة *Method call* بجسم الطريقة المطلوب *Method Body*.

وعندما تتم عملية الربط قبل تنفيذ البرنامج (من خلال المترجم أو الرابط) نسميها إذا بالربط المبكر *Early Binding*.

ربما لم تسمع من قبل عن هذا الربط المبكر لأنه لم يكن ضمن أحد الخيارات الموجودة في البرمجة الإجرائية *Procedural Programming*. فمترجمات لغة C مثلاً تمتلك نوعاً واحداً من استدعاء الطرق وهو الربط المبكر.

في البرنامج السابق توجد مشكلة تتعلق بالربط المبكر لأن المترجم لا يستطيع معرفة الطريقة الصحيحة المطلوب استدعاءها عندما يوجد مؤشر *Instrument* واحد فقط. الحل هنا هو إجراء الربط المتأخر *late binding* حيث تتم عملية الربط أثناء وقت التنفيذ *run-time* وذلك بالاعتماد على نمط العنصر.

يسمى هذا الربط أيضاً بالربط الديناميكي *dynamic binding* أو الربط أثناء وقت التنفيذ *run-time binding*.

وعندما يتم استخدام هذا النوع من الربط، يجب توفر ميكانيكية تحدد نمط العنصر أثناء وقت التنفيذ ومن ثم استدعاء الطريقة المناسبة.

وفي لغة جافا، تستخدم جميع الطرق الربط المتأخر *late binding* إلا في حال التصريح عن طريقة بأنها نهائية *final*. لذلك لن تحتاج أبداً إلى اتخاذ قرار بشأن استخدام الربط المتأخر لأنه سيتم بشكل تلقائي.



من فضلك أعطني مثلاً يوضح لي ذلك...

سنأخذ هنا المثال التقليدي البسيط المتعلق بالأشكال *shape*. كما ذكرنا سابقاً هناك صف أساسي اسمه *Shape* والعديد من الصفوف المشتقة هي: *Circle* و *Square* و *Triangle* وغيرها.

توضح التعليلة البسيطة التالية كيفية القيام بالتوجيه للأعلى *Upcast*:

```
Shape s = new Circle();
```

تلاحظ هنا بأنه يتم إنشاء عنصر *Circle* جديد، أما المؤشر الناتج فيدل مباشرة على *Shape*. الآن عندما تقوم باستدعاء أحد طرق الصف الأساسي (والمهيمنة *overridden* في الصفوف المشتقة) مثلاً:

```
s.draw();
```

ستتوقع هنا أن الطريقة *draw()* الموجودة في الصف *Shape* هي الطريقة التي سيتم استدعاؤها. لكن سيخيب ظنك وسيتم تنفيذ الطريقة *draw()* الموجودة في الصف *Circle* نتيجة للربط المتأخر *late binding*.

الصفوف والطرق المجردة Abstract

...Classes and Methods

تستخدم الصفوف المجردة عند الحاجة للتعامل مع مجموعة من الصفوف من خلال واجهة مشتركة *common interface*. ويتم استدعاء جميع الصفوف المشتقة من الصف المجرد باستخدام تقنية الربط الديناميكي *dynamic binding*.

وإذا كان لديك صف مجرد كالصف *Instrument* مثلاً، فإن جميع عناصر هذا الصف لن تحمل أي معنى. السبب في ذلك هو أن الهدف من الصف *Instrument* هو التعبير عن الواجهة فقط، وليس لإعطاء أي عمل خاص له، لذلك يفضل عدم إنشاء أية عناصر في الصف المجرد ومنع المستخدم من القيام بذلك، وذلك بجعل جميع الطرق في الصف *Instrument* تطبع رسائل خطأ مثلاً.

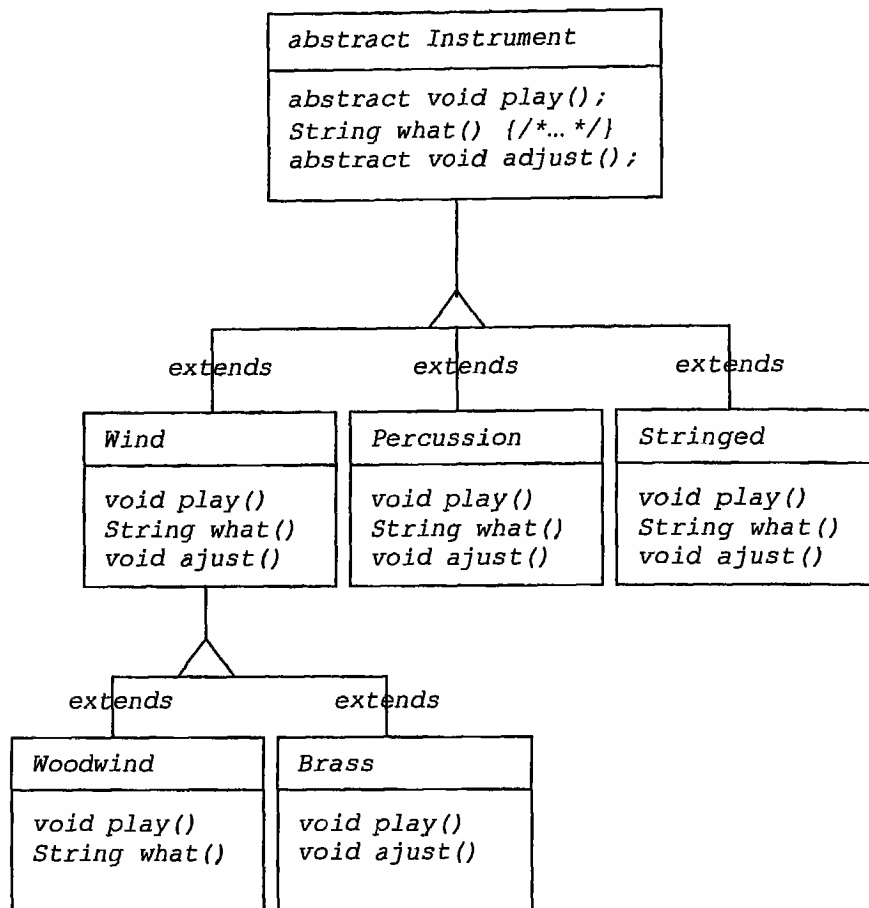
المشكلة في الحل السابق هو أن معرفة إنشاء عناصر الصف المجرد ستتأخر حتى زمن التنفيذ *run time*، لذلك أضافت جافا ما يسمى بالطرق المجردة *Abstract Methods* من أجل إظهار الأخطاء أثناء زمن الترجمة *compile time*. والطرق المجردة عبارة عن طرق غير مكتملة تمتلك توصيفاً فقط وليس لها أي جسم، وتأخذ الشكل التالي:

```
abstract void X();
```

عندما تقوم بالتوريث من صف مجرد وترغب بإنشاء عناصر في الصف المشتق، فإن عليك تعريف جميع الطرق المجردة في الصف الأساسي، وإلا فإن المترجم سيعتبرها أيضاً طرقاً مجردة وسيجبرك على إضافة كلمة المفتاح *abstract* للصف الجديد. طبعاً بالإمكان إنشاء صف مجرد دون إنشاء طرقاً مجردة فيه لسد الطريق على إنشاء ممثلين لهذا الصف.

يمكن كمثال تحويل الصف *Instrument* إلى صف مجرد. ويمكن جعل بعض الطرق في هذا الصف مجردة إذ أنه ليس من الضروري أن تكون جميع الطرق مجردة. انظر معي إلى المخطط التالي:





أما البرنامج التالي فهو نسخة معدلة من البرنامج الأول في هذا الفصل حيث قمنا هنا باستخدام الصفوف والطرق المجردة:

```

//: Music4.java
// Abstract classes and methods
import java.util.*;
abstract class Instrument4 {
    int i; // storage allocated for each
    public abstract void play();
    public String what() {
        return "Instrument4";
    }
}
  
```

```

    public abstract void adjust();
}
class Wind4 extends Instrument4 {
    public void play() {
        System.out.println("Wind4.play()");
    }
    public String what() { return "Wind4"; }
    public void adjust() {}
}
class Percussion4 extends Instrument4 {
    public void play() {
        System.out.println("Percussion4.play()");
    }
    public String what() { return "Percussion4"; }
    public void adjust() {}
}
class Stringed4 extends Instrument4 {
    public void play() {
        System.out.println("Stringed4.play()");
    }
    public String what() { return "Stringed4"; }
    public void adjust() {}
}
class Brass4 extends Wind4 {
    public void play() {
        System.out.println("Brass4.play()");
    }
    public void adjust() {
        System.out.println("Brass4.adjust()");
    }
}
class Woodwind4 extends Wind4 {
    public void play() {
        System.out.println("Woodwind4.play()");
    }
    public String what() { return "Woodwind4"; }
}

```



```

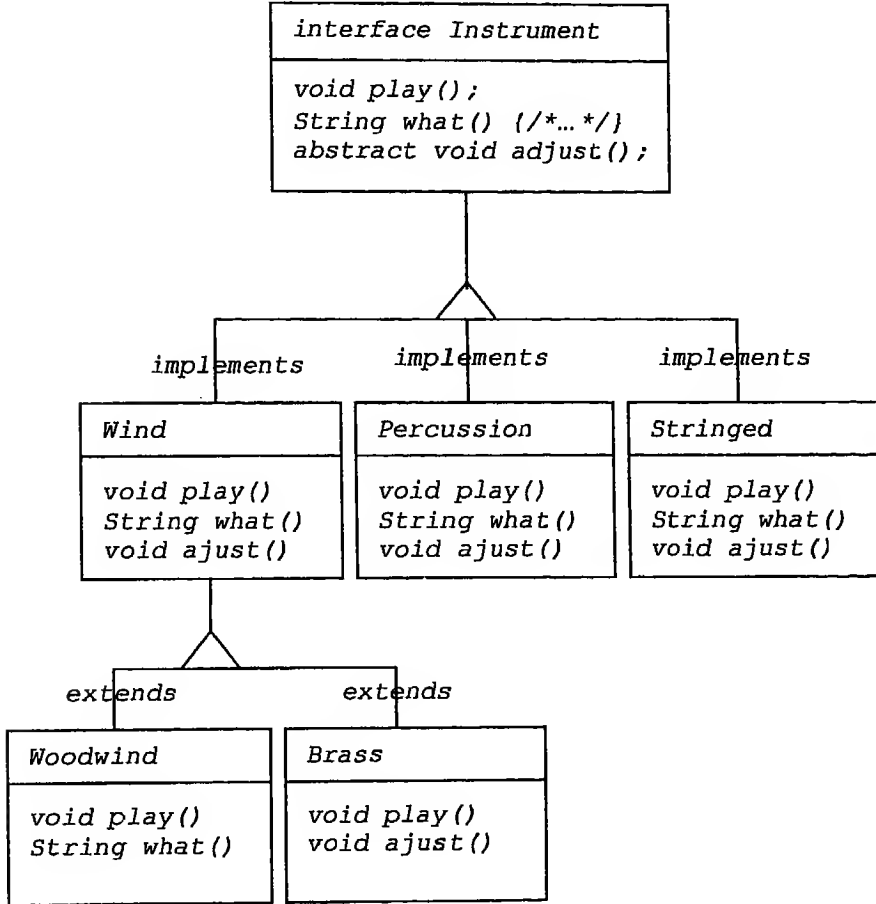
public class Music4 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument4 i) {
        // ...
        i.play();
    }
    static void tuneAll(Instrument4[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }
    public static void main(String[] args) {
        Instrument4[] orchestra = new
        Instrument4[5];
        int i = 0;
        // Upcasting during addition to the array:
        orchestra[i++] = new Wind4();
        orchestra[i++] = new Percussion4();
        orchestra[i++] = new Stringed4();
        orchestra[i++] = new Brass4();
        orchestra[i++] = new Woodwind4();
        tuneAll(orchestra);
    }
} ///:~

```

يوجد دور للصفوف والطرق المجردة في بناء الواجهة...

تفيد كلمة المفتاح *interface* بأخذ مفهوم التجريد كخطوة للأمام. حيث يمكنك اعتبار الواجهة كصف مجرد صاف تماماً، فهي تسمح بإنشاء نموذج عن الصف حيث تحتوي على أسماء الطرق وقوائم الوسائط وأنماط الإرجاع إلا أنها لا تحتوي على أجسام الطرق. ويمكن للواجهة *interface* أن تحتوي على أعضاء المعطيات *data members* الأنماط الأولية *primitive types*. باختصار فإن الواجهة *interface* هي فقط نموذج *form* دون أي عمل *implementation*.

لإنشاء واجهة `interface` قم باستخدام كلمة المفتاح `interface` بدلا من `class`، ولإعطاء دور لهذه الواجهة استخدم كلمة المفتاح `implements`، في هذه الحالة كأنك تقول: "الواجهة `interface` هي ما أرغب بإظهاره لك، وإليك الآن كيفية عملها". سنعدل الآن المخطط السابق ليصبح على الشكل:



وسنقوم هنا بإنشاء نسخة معدلة عن البرنامج السابق لتبيان أهمية استخدام `interface` و `implements`:

```
//: Music5.java
// Interfaces
```



```

import java.util.*;
interface Instrument5 {
    // Compile-time constant:
    int i = 5; // static & final
    // Cannot have method definitions:
    void play(); // Automatically public
    String what();
    void adjust();
}
class Wind5 implements Instrument5 {
    public void play() {
        System.out.println("Wind5.play()");
    }
    public String what() { return "Wind5"; }
    public void adjust() {}
}
class Percussion5 implements Instrument5 {
    public void play() {
        System.out.println("Percussion5.play()");
    }
    public String what() { return "Percussion5"; }
    public void adjust() {}
}
class Stringed5 implements Instrument5 {
    public void play() {
        System.out.println("Stringed5.play()");
    }
    public String what() { return "Stringed5"; }
    public void adjust() {}
}
class Brass5 extends Wind5 {
    public void play() {
        System.out.println("Brass5.play()");
    }
    public void adjust() {
        System.out.println("Brass5.adjust()");
    }
}

```

```

}
class Woodwind5 extends Wind5 {
    public void play() {
        System.out.println("Woodwind5.play()");
    }
    public String what() { return "Woodwind5"; }
}
public class Music5 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument5 i) {
        // ...
        i.play();
    }
    static void tuneAll(Instrument5[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }
    public static void main(String[] args) {
        Instrument5[] orchestra = new
        Instrument5[5];
        int i = 0;
        // Upcasting during addition to the array:
        orchestra[i++] = new Wind5();
        orchestra[i++] = new Percussion5();
        orchestra[i++] = new Stringed5();
        orchestra[i++] = new Brass5();
        orchestra[i++] = new Woodwind5();
        tuneAll(orchestra);
    }
} ///:~

```



التوريث المتعدد *Multiple Inheritance*...

في جافا كما في C++ تستطيع إجراء التوريث المتعدد.

لكن ما هو التوريث المتعدد؟

كما ذكرنا سابقا فإن الواجهة *Interface* هي ببساطة صف مجرد ليس لها عمل تنفيذي *implementation*. لذلك فهي لا تأخذ مساحة تخزين، وبالتالي فلا شيء يمنع من أن يتم دمج عدة واجهات.

يسمى الدمج المتعدد للواجهات في لغة C++ بالتوريث المتعدد *multiple inheritance* والذي كان بسبب العديد من الإشكاليات لأن أي صف يمكنه أن يمتلك عملا تنفيذيا *implementation*.

أما لغة جافا فلقد سمحت بتعدد التوريث، إلا أنها ابتعدت عن المشاكل التي كانت تواجه C++ لأنها لم تسمح إلا لصف واحد من الصفوف أن يمتلك عملا تنفيذيا. وفي الصفوف المشتقة من صف أساسي غير تجريدي فلا تستطيع التوريث إلا من صف واحد فقط.

يوضح المثال التالي صفا أساسيا مرتبطا بعدة واجهات لتوليد صف جديد:

```
//: Adventure.java
// Multiple interfaces
import java.util.*;
interface CanFight {
    void fight();
}
interface CanSwim {
    void swim();
}
interface CanFly {
    void fly();
}
class ActionCharacter {
    public void fight() {}
}
```

```

}
class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
    public void swim() {}
    public void fly() {}
}
public class Adventure {
    static void t(CanFight x) { x.fight(); }
    static void u(CanSwim x) { x.swim(); }
    static void v(CanFly x) { x.fly(); }
    static void w(ActionCharacter x) { x.fight(); }
}
public static void main(String[] args) {
    Hero i = new Hero();
    t(i); // Treat it as a CanFight
    u(i); // Treat it as a CanSwim
    v(i); // Treat it as a CanFly
    w(i); // Treat it as an ActionCharacter
}
} ///:~

```

يمكنك ملاحظة كيف أن العنصر *Hero* يقوم بدمج الصف الفعلي *ActionCharacter* في عدة واجهات هي: *CanFight* و *CanSwim* و *CanFly*. في هذه الحالة يجب تحديد الصف الفعلي أولاً ثم تليه الواجهات. أما في الصف *Adventure* فيمكنك ملاحظة وجود أربع طرق تأخذ الواجهات المختلفة كوسائط إضافة إلى الصف الفعلي. لذلك عندما يتم إنشاء العنصر *Hero* يمكن تمريره إلى أي من هذه الطرق، أي يمكن توجيهه للأعلى *upcast* من أجل كل واجهة من الواجهات.



يمكنك توسيع واجهتك باستخدام التوريث...

تستطيع ببساطة إضافة توصيف طريقة جديدة لواجهة `interface` باستخدام التوريث، كذلك بإمكانك دمج عدة واجهات لإعطاء واجهة جديدة باستخدام التوريث. ستحصل في الحالتين على واجهة جديدة كما في المثال التالي:

```
//: HorrorShow.java
// Extending an interface with inheritance
interface Monster {
    void menace();
}
interface DangerousMonster extends Monster {
    void destroy();
}
interface Lethal {
    void kill();
}
class DragonZilla implements DangerousMonster {
    public void menace() {}
    public void destroy() {}
}
interface Vampire
extends DangerousMonster, Lethal {
    void drinkBlood();
}
class HorrorShow {
    static void u(Monster b) { b.menace(); }
    static void v(DangerousMonster d) {
        d.menace();
        d.destroy();
    }
}
public static void main(String[] args) {
    DragonZilla if2 = new DragonZilla();
    u(if2);
    v(if2);
}
```

```
} ///:~
```

بإمكانك أيضا استخدام الواجهات لإنشاء مجموعات من الثوابت...

باعتبار أن أي حقل تضعه في واجهة ما يكون تلقائيا إما *static* أو *final*، لذلك فإن الواجهة أداة ممتازة لإنشاء مجموعات الثوابت تماما كالنمط *enum* المستخدم في C أو C++. يوضح المثال التالي كيفية استخدام الواجهات لتجميع الثوابت:

```
//: Months.java
// Using interfaces to create groups of
// constants
package c07;
public interface Months {
    int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
} ///:~
```

الصفوف الداخلية Inner classes...

نستطيع في لغة جافا تعريف صف ضمن صف آخر، وهو ما نسميه بالصف الداخلي *inner class*. وتفيد الصفوف الداخلية في تجميع الصفوف المرتبطة منطقيا. ومن المهم فهمه أيضا أن الصفوف الداخلية تختلف عن الصفوف المركبة *composition classes*. وكمثال على الصفوف الداخلية سنأخذ البرنامج التالي:

```
//: Parcell.java
// Creating inner classes
package c07.parcell;
```



```

public class Parcel1 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    // Using inner classes looks just like
    // using any other class, within Parcel1:
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Tanzania");
    }
} ///:~

```

إذا كما تلاحظ فإن الصفوف الداخلية تفيد في إخفاء الأسماء *name-hiding*، كذلك في تنظيم الترميز *code-organization*. أيضا عندما تقوم بإنشاء صف داخلي، فإن عناصر هذا الصف ترتبط مع العناصر المطوقة *enclosed objects* التي أنشأتها، لذلك تستطيع الوصول إلى أعضاء العناصر المطوقة *enclosed objects* دون أن تمتلك أي امتياز. بالإضافة إلى ذلك فإن الصفوف الداخلية تمتلك حقوق الوصول إلى جميع عناصر الصف المطوقة *enclosed objects*. هذا الأمر يختلف تماما عن الصفوف المتداخلة *nested classes* في C++، فهي عبارة عن تقنية لإخفاء الأسماء فقط و لا يوجد أي ارتباط مع العناصر المطوقة *enclosed objects*.

والمثال التالي يوضح ما سبق أن شرحناه:

```

//: Sequence.java
// Holds a sequence of Objects
interface Selector {
    boolean end();
    Object current();
    void next();
}
public class Sequence {
    private Object[] o;
    private int next = 0;
    public Sequence(int size) {
        o = new Object[size];
    }
    public void add(Object x) {
        if(next < o.length) {
            o[next] = x;
            next++;
        }
    }
    private class SSelector implements Selector {
        int i = 0;
        public boolean end() {
            return i == o.length;
        }
        public Object current() {
            return o[i];
        }
        public void next() {
            if(i < o.length) i++;
        }
    }
    public Selector getSelector() {
        return new SSelector();
    }
    public static void main(String[] args) {
        Sequence s = new Sequence(10);
        for(int i = 0; i < 10; i++)
            s.add(Integer.toString(i));
    }
}

```



```

Selector sl = s.getSelector();
while(!sl.end()) {
    System.out.println((String)sl.current());
    sl.next();
}
}
} ///:~

```

الصف *Sequence* هو ببساطة عبارة عن مصفوفة ثابتة الحجم من عناصر *Object*. ويمكنك استدعاء الطريقة *add()* لإضافة عنصر *Object* جديد في نهاية السلسلة *Sequence*. ومن أجل جلب أي من العناصر في *Sequence*، توجد واجهة بالاسم *Selector* تسمح بمعرفة إن كنا في نهاية السلسلة *end()*، أو للبحث عن العنصر الحالي *current()*، أو للانتقال إلى العنصر التالي *next()* في السلسلة.

وعلى اعتبار أن *Selector* عبارة عن واجهة *interface*، فهذا يمكن العديد من الصفوف الأخرى من تنفيذ هذه الواجهة بطريقتها الخاصة. كذلك يمكن للعديد من الطرق أخذ *interface* كوسيط من أجل إنشاء ترميز شامل. فمثلاً الصف *Sselector* عبارة عن صف خاص يحدد عمل طرق الصف *Selector*.

في الطريقة *main()* يتم أولاً إنشاء عنصر *Sequence*، ثم إضافة عدد من عناصر *string*. يتم بعد ذلك توليد *Selector* بطلب الطريقة *getSelector()* والذي يستخدم للتنقل في *Selector* واختيار كل منها. لذلك كما ترى يمكن للصف الداخلي الوصول إلى أعضاء الصف المطوق *enclosing class*، لأنه يحتفظ بمؤشر للعنصر في الصف المطوق الذي قام بإنشائه.

البانيات وتعددية الأشكال...

يتم دوما استدعاء باني الصف الأساسي ضمن باني كل صف مشتق. ويجب أن تستدعي جميع البانيات، وإلا فإن العنصر لن يبنى بشكل سليم. لذلك فإن المترجم يجبر استدعاء الباني إلى كل جزء في الصف المشتق.

لنأخذ مثالا يوضح أثر التركيب والتوريث وتعددية الأشكال في ترتيب الباني:

```
//: Sandwich.java
// Order of constructor calls
class Meal {
    Meal() { System.out.println("Meal()"); }
}
class Bread {
    Bread() { System.out.println("Bread()"); }
}
class Cheese {
    Cheese() { System.out.println("Cheese()"); }
}
class Lettuce {
    Lettuce() { System.out.println("Lettuce()"); }
}
class Lunch extends Meal {
    Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch {
    PortableLunch() {
        System.out.println("PortableLunch()");
    }
}
class Sandwich extends PortableLunch {
    Bread b = new Bread();
    Cheese c = new Cheese();
    Lettuce l = new Lettuce();
    Sandwich() {
        System.out.println("Sandwich()");
    }
}
```



```

}
public static void main(String[] args) {
    new Sandwich();
}
} ///:~

```

نقوم في هذا المثال بإنشاء صف معقد خارج بقية الصفوف، ولكل صف بان خاص. الصف الهام هنا هو *Sandwich* والذي يعبر عن ثلاثة مستويات من التوريث (أو أربعة إذا اعتبرنا أن التوريث الداخلي من *Object*)، كذلك ثلاثة عناصر أعضاء. وعندما يتم إنشاء عنصر *Sandwich* في *main()* سيظهر خرج البرنامج على الشكل:

```

Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()

```

هذا يعني أن ترتيب استدعاء الباني لعنصر مركب يتم كما يلي:

١. يستدعى أولاً باني الصف الأساسي. وتكرر هذه الخطوة طالما أنه يتم بناء الجذر أولاً، يتبع ذلك الصف المشتق التالي حتى الوصول إلى آخر صف مشتق.
٢. يتم استدعاء المحددات الابتدائية للأعضاء بترتيب توصيفها.
٣. أخيراً يتم استدعاء باني جسم الصف المشتق.

التوريث والطريقة *finalize()* ...

كما رأينا سابقاً، فإنك عندما تستخدم طريقة التركيب *composition* لإنشاء صف جديد، فلن تقلق أبداً بالنسبة لعملية إنهاء عناصر الأعضاء ضمن هذا الصف. أما عند التوريث فإن عليك الهيمنة *override* على الطريقة *finalize()* المعرفة في الصف المشتق. لكن تذكر أن تقوم باستدعاء *finalize()* الموجودة في الصف الأساسي وإلا فإن عملية الإنهاء لن تتم بشكل صحيح في الصف الأساسي.

والمثال التالي يوضح مذكرناه:

```

//: Frog.java
// Testing finalize with inheritance
class DoBaseFinalization {
    public static boolean flag = false;
}
class Characteristic {
    String s;
    Characteristic(String c) {
        s = c;
        System.out.println(
            "Creating Characteristic " + s);
    }
    protected void finalize() {
        System.out.println(
            "finalizing Characteristic " + s);
    }
}
class LivingCreature {
    Characteristic p =
        new Characteristic("is alive");
    LivingCreature() {
        System.out.println("LivingCreature()");
    }
    protected void finalize() {
        System.out.println(
            "LivingCreature finalize");
        // Call base-class version LAST!
        if(DoBaseFinalization.flag)
            try {
                super.finalize();
            } catch(Throwable t) {}
    }
}
class Animal extends LivingCreature {
    Characteristic p =
        new Characteristic("has heart");
}

```



```

Animal() {
    System.out.println("Animal()");
}
protected void finalize() {
    System.out.println("Animal finalize");
    if(DoBaseFinalization.flag)
        try {
            super.finalize();
        } catch(Throwable t) {}
}
}
class Amphibian extends Animal {
    Characteristic p =
    new Characteristic("can live in water");
    Amphibian() {
        System.out.println("Amphibian()");
    }
    protected void finalize() {
        System.out.println("Amphibian finalize");
        if(DoBaseFinalization.flag)
            try {
                super.finalize();
            } catch(Throwable t) {}
    }
}
public class Frog extends Amphibian {
    Frog() {
        System.out.println("Frog()");
    }
    protected void finalize() {
        System.out.println("Frog finalize");
        if(DoBaseFinalization.flag)
            try {
                super.finalize();
            } catch(Throwable t) {}
    }
}
public static void main(String[] args) {
    if(args.length != 0 &&

```

```

        args[0].equals("finalize"))
            DoBaseFinalization.flag = true;
    else
        System.out.println("not finalizing
bases");
    new Frog(); // Instantly becomes garbage
    System.out.println("bye!");
    // Must do this to guarantee that all
    // finalizers will be called:
    System.runFinalizersOnExit(true);
}
} ///:~

```

سيظهر خرج البرنامج السابق على الشكل:

```

not finalizing bases
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water

```





المشكلة الأكبر التي تعترضك في أغلب برامجك هي أنك، وفي كثير من الأحيان، بحاجة لإنشاء أي عدد من العناصر في أي وقت وفي أي مكان.

ولحل هذه المشكلة نقوم جافا باستخدام طرق عديدة لاحتواء العناصر (أو حتى مؤشرات العناصر) كالمصفوفات التي تحدثنا عنها سابقا، وسنتحدث عنها بالتفصيل لاحقا. تحتوي كذلك مكتبة أدوات جافا على بعض صفوف المجموعات *collection classes*

والتي منسجمها المجموعات *collection* اختصاراً، وهي تساعد على حمل و معالجة عناصرك.

المصفوفات *Arrays*...

قمنا في الفصول السابقة بتعليمك كيفية القيام بتعريف مصفوفة، كذلك شرحنا كيفية تحديد القيم الابتدائية لها. أما الآن فسنبين لك كيف يمكن لهذه المصفوفات احتواء عناصرك. تعتبر المصفوفات من أهم الطرق المستخدمة لاحتواء العناصر، وتتميز عن المجموعات من ناحية النمط والفعالية.

وبغض النظر عن نمط المصفوفة التي تتعامل معها، فإنّ محدّد المصفوفة عبارة عن مؤشر لعنصر فعلي تمّ إنشاؤه على الكومة *heap*. ويمكن إنشاء عنصر الكومة إما بشكل ضمني *implicit* كجزء من عملية تحديد القيمة الابتدائية للمصفوفة، أو بشكل صريح *explicit* باستخدام تعليمة *new*. ويستخدم العضو *length* (وهو في الواقع الحقل الوحيد المتاح أو الطريقة الوحيدة المتاحة) لتحديد عدد العناصر التي يمكنك تخزينها في المصفوفة.

يوضح المثال التالي كيفية تعريف مصفوفة، وتحديد القيم الابتدائية لها، وكذلك كيفية التعامل معها:

```
//: ArraySize.java
// Initialization & re-assignment of arrays
package c08;
class Weeble {} // A small mythical creature
public class ArraySize {
    public static void main(String[] args) {
        // Arrays of objects:
        Weeble[] a; // Null handle
        Weeble[] b = new Weeble[5]; // Null handles
        Weeble[] c = new Weeble[4];
        for(int i = 0; i < c.length; i++)
            c[i] = new Weeble();
```



```

Weeble[] d = {
    new Weeble(), new Weeble(), new Weeble()
};
// Compile error: variable a not
// initialized:
//!System.out.println("a.length=" +
// a.length);
System.out.println("b.length = " +
b.length);
// The handles inside the array are
// automatically initialized to null:
for(int i = 0; i < b.length; i++)
    System.out.println("b[" + i + "]= " +
b[i]);
System.out.println("c.length = " +
c.length);
System.out.println("d.length = " +
d.length);
a = d;
System.out.println("a.length = " +
a.length);
// Java 1.1 initialization syntax:
a = new Weeble[] {
    new Weeble(), new Weeble()
};
System.out.println("a.length = " +
a.length);
// Arrays of primitives:
int[] e; // Null handle
int[] f = new int[5];
int[] g = new int[4];
for(int i = 0; i < g.length; i++)
    g[i] = i*i;
int[] h = { 11, 47, 93 };
// Compile error: variable e not
// initialized:
//!System.out.println("e.length=" +
// e.length);

```

```

System.out.println("f.length = " +
f.length);
// The primitives inside the array are
// automatically initialized to zero:
for(int i = 0; i < f.length; i++)
    System.out.println("f[" + i + "]= " +
f[i]);
System.out.println("g.length = " +
g.length);
System.out.println("h.length      =      "      +
h.length);
e = h;
System.out.println("e.length = " +
e.length);
// Java 1.1 initialization syntax:
e = new int[] { 1, 2 };
System.out.println("e.length = " +
e.length);
}
} ///:~

```

أما خرج هذا البرنامج فسيكون على الشكل:

```

b.length = 5
b[0]=null
b[1]=null
b[2]=null
b[3]=null
b[4]=null
c.length = 4
d.length = 3
a.length = 3
a.length = 2
f.length = 5
f[0]=0
f[1]=0
f[2]=0
f[3]=0
f[4]=0

```



```
g.length = 4
h.length = 3
e.length = 3
e.length = 2
```

هل تسمح لي جافا بإرجاع مصفوفة!!!؟

من أهم الصعوبات التي كانت تواجهك في لغة C أو C++ هي عدم إمكانية إرجاع مصفوفة، وإنما إرجاع مؤشر لهذه المصفوفة. لذلك كنت دوما بحاجة لمعرفة زمن استخدام المصفوفة حتى لا تسبب ضعفا للذاكرة.

جافا تقوم بنفس العمل، لكن مع جافا لن تقلق أبدا من التعامل مع المصفوفات، لأن مجمع النفايات *garbage collector* سيكون في الجوار وسيظف كل مصفوفة كسولة. لنأخذ المثال التالي الذي يوضح كيفية إرجاع مصفوفة سلاسل `:String`

```
//: IceCream.java
// Returning arrays from methods
public class IceCream {
    static String[] flav = {
        "Chocolate", "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    static String[] flavorSet(int n) {
        // Force it to be positive & within bounds:
        n = Math.abs(n) % (flav.length + 1);
        String[] results = new String[n];
        int[] picks = new int[n];
        for(int i = 0; i < picks.length; i++)
            picks[i] = -1;
        for(int i = 0; i < picks.length; i++) {
            retry:
            while(true) {
                int t =
                    (int) (Math.random() * flav.length);
```

```

        for(int j = 0; j < i; j++)
            if(picks[j] == t) continue retry;
            picks[i] = t;
            results[i] = flav[t];
            break;
        }
    }
    return results;
}

public static void main(String[] args) {
    for(int i = 0; i < 20; i++) {
        System.out.println(
            "flavorSet(" + i + ") = ";
        String[] fl = flavorSet(flav.length);
        for(int j = 0; j < fl.length; j++)
            System.out.println("\t" + fl[j]);
        }
    }
} ///:~

```



المجموعات *Collections*؟

كما رأينا سابقا فإن المصفوفات هي الخيار الأفضل عندما تحتاج إلى التعامل مع مجموعة من العناصر الأولية. لكن عندما لا تستطيع معرفة عدد العناصر التي ستحتاجها، أو عندما تكون بحاجة إلى التعامل مع عناصر بأنماط غير أولية، فأنت إذا بحاجة للعمل مع المجموعات *Collections*.

تساعدك جافا على التعامل مع أربعة أنماط من صفوف المجموعات وهي: *Vector* و *BitSet* و *Stack* و *Hashtable*.

لكن انتبه، فأنت ستتعامل مع نمط غير معروف!!؟

من أهم السليبيات التي تواجهها عند استخدامك لمجموعات جافا، فقدانك لنمط المعلومات عندما تقوم بوضع عنصر في مجموعة. والسبب في ذلك هو أن مبرمج المجموعة لا يمتلك أية فكرة عن النمط المحدد للعناصر التي ستضعها في المجموعة. لذلك فلن جعل المجموعات تتعامل مع نمط محدد فقط سوف يفقدها الهدف الأساسي من إنشائها. من أجل ذلك تقوم المجموعات بالتعامل مع مؤشرات لعناصر من نمط *Object*، والتي هي بالطبع أي عنصر ضمن جافا.

والآن سنوضح كل ما ذكرناه في المثال التالي:

```
//: CatsAndDogs.java
// Simple collection example (Vector)
import java.util.*;
class Cat {
    private int catNumber;
    Cat(int i) {
        catNumber = i;
    }
    void print() {
        System.out.println("Cat #" + catNumber);
    }
}
```

سلسلة الرضا للمعلومات

```

}
}
class Dog {
    private int dogNumber;
    Dog(int i) {
        dogNumber = i;
    }
    void print() {
        System.out.println("Dog #" + dogNumber);
    }
}
public class CatsAndDogs {
    public static void main(String[] args) {
        Vector cats = new Vector();
        for(int i = 0; i < 7; i++)
            cats.addElement(new Cat(i));
        // Not a problem to add a dog to cats:
        cats.addElement(new Dog(7));
        for(int i = 0; i < cats.size(); i++)
            ((Cat)cats.elementAt(i)).print();
        // Dog is detected only at run-time
    }
} //:~

```

هناك نوع من العدادات اسمه **Enumeration**...

العداد عبارة عن عنصر يفيدك في التنقل ضمن سلسلة عناصر بالإضافة إلى اختيار كل عنصر في هذه السلسلة دون أخذ بنية هذه السلسلة بعين الاعتبار.

ويعتبر العداد *Enumeration* أحد العدادات المستخدمة في لغة جافا، والمثال التالي يوضح كيفية عمله:

```

//: CatsAndDogs2.java
// Simple collection with Enumeration
import java.util.*;
class Cat2 {
    private int catNumber;

```




```

Cat2(int i) {
    catNumber = i;
}
void print() {
    System.out.println("Cat number "
+catNumber);
}
}
class Dog2 {
    private int dogNumber;
    Dog2(int i) {
        dogNumber = i;
    }
    void print() {
        System.out.println("Dog number "
+dogNumber);
    }
}
public class CatsAndDogs2 {
    public static void main(String[] args) {
        Vector cats = new Vector();
        for(int i = 0; i < 7; i++)
            cats.addElement(new Cat2(i));
        // Not a problem to add a dog to cats:
        cats.addElement(new Dog2(7));
        Enumeration e = cats.elements();
        while(e.hasMoreElements())
            ((Cat2)e.nextElement()).print();
        // Dog is detected only at run-time
    }
} //::~~

```

ما يمكن ملاحظته هو أن التغيير الوحيد يكون في الأسطر الأخيرة، فبدلاً من كتابة:

```

for(int i = 0; i < cats.size(); i++)
    ((Cat)cats.elementAt(i)).print();

```

يستخدم العداد Enumeration للتنقل ضمن السلسلة:

```

while(e.hasMoreElements())
    ((Cat2)e.nextElement()).print();

```

سلسلة الرضا للمعلومات

سنتحدث الآن عن أنواع المجموعات الأربع بالتفصيل:

المجموعة الأبسط *Vector*...

وهو أبسط الأنماط استخداماً، وستحتاج في أغلب الأحيان إلى *addElement()* لإدراج العناصر، و *elementAll()* للحصول عليها واحداً تلو الآخر، و *element()* لجلب عداد *Enumeration* إلى السلسلة، إضافة إلى العديد من الطرق الأخرى التي يمكنك استخدامها. ولقد رأينا في البرنامجين السابقين كيفية استخدام هذا النمط.

يوجد نمط آخر قريب من *Vector* هو *BitSet*...

وهو عبارة عن شعاع *Vector* من البتات *bits*. ويستخدم هذا النمط عند الحاجة لتخزين الكثير من المعلومات ذات النمط *on/off*. يوضح البرنامج التالي كيفية استخدام هذا النمط:

```
//: Bits.java
// Demonstration of BitSet
import java.util.*;
public class Bits {
    public static void main(String[] args) {
        Random rand = new Random();
        // Take the LSB of nextInt():
        byte bt = (byte)rand.nextInt();
        BitSet bb = new BitSet();
        for(int i = 7; i >= 0; i--)
            if(((1 << i) & bt) != 0)
                bb.set(i);
            else
                bb.clear(i);
        System.out.println("byte value: " + bt);
        printBitSet(bb);
        short st = (short)rand.nextInt();
        BitSet bs = new BitSet();
```



```

for(int i = 15; i >=0; i--)
    if(((1 << i) & st) != 0)
        bs.set(i);
    else
        bs.clear(i);
System.out.println("short value: " + st);
printBitSet(bs);
int it = rand.nextInt();
BitSet bi = new BitSet();
for(int i = 31; i >=0; i--)
    if(((1 << i) & it) != 0)
        bi.set(i);
    else
        bi.clear(i);
System.out.println("int value: " + it);
printBitSet(bi);
// Test bitsets >= 64 bits:
BitSet b127 = new BitSet();
b127.set(127);
System.out.println("set bit 127: " + b127);
BitSet b255 = new BitSet(65);
b255.set(255);
System.out.println("set bit 255: " + b255);
BitSet b1023 = new BitSet(512);
// Without the following, an exception is
// thrown
// in the Java 1.0 implementation of
// BitSet:
// b1023.set(1023);
b1023.set(1024);
System.out.println("set bit 1023: " +
b1023);
}
static void printBitSet(BitSet b) {
System.out.println("bits: " + b);
String bbits = new String();
for(int j = 0; j < b.size() ; j++)
    bbits += (b.get(j) ? "1" : "0");

```

```

System.out.println("bit pattern: " +
    bbits);
}
} ///:~

```

المكدس Stack...

ويسمى عادة *last-in, first out* أو مجموعة *LIFO*، أي ما يدخل أخيراً يخرج أولاً. ويرث هذا النمط جميع خصائص النمط *Vector* إضافة إلى بعض الخصائص المميزة لهذا النمط. كمثال على استخدام هذا النمط:

```

//: Stacks.java
// Demonstration of Stack Class
import java.util.*;
public class Stacks {
    static String[] months = {
        "January", "February", "March", "April",
        "May", "June", "July", "August",
        "September",
        "October", "November", "December" };
    public static void main(String[] args) {
        Stack stk = new Stack();
        for(int i = 0; i < months.length; i++)
            stk.push(months[i] + " ");
        System.out.println("stk = " + stk);
        // Treating a stack as a Vector:
        stk.addElement("The last line");
        System.out.println(
            "element 5 = " + stk.elementAt(5));
        System.out.println("popping elements:");
        while(!stk.empty())
            System.out.println(stk.pop());
    }
} ///:~

```



النمط الرابع والأخير هو `hashtable`...

كما رأينا سابقا فإن النمط `Vector` يسمح بالاختيار من سلسلة عناصر من خلال استخدام عدد، أي يتم ربط الأرقام بالعناصر. لكن بالإمكان استخدام معيار آخر للاختيار كالاختيار من سلسلة مثلا كما في الصف المجرد `Dictionary`. يوضح المثال التالي كيفية استخدام هذا النمط:

```
//: AssocArray.java
// Simple version of a Dictionary
import java.util.*;
public class AssocArray extends Dictionary {
    private Vector keys = new Vector();
    private Vector values = new Vector();
    public int size() { return keys.size(); }
    public boolean isEmpty() {
        return keys.isEmpty();
    }
    public Object put(Object key, Object value) {
        keys.addElement(key);
        values.addElement(value);
        return key;
    }
    public Object get(Object key) {
        int index = keys.indexOf(key);
        // indexOf() Returns -1 if key not found:
        if(index == -1) return null;
        return values.elementAt(index);
    }
    public Object remove(Object key) {
        int index = keys.indexOf(key);
        if(index == -1) return null;
        keys.removeElementAt(index);
        Object returnval = values.elementAt(index);
        values.removeElementAt(index);
        return returnval;
    }
    public Enumeration keys() {
```

سلسلة الرضا للمعلومات

```

return keys.elements();
}
public Enumeration elements() {
    return values.elements();
}
// Test it:
public static void main(String[] args) {
    AssocArray aa = new AssocArray();
    for(char c = 'a'; c <= 'z'; c++)
        aa.put(String.valueOf(c),
            String.valueOf(c).toUpperCase());
    char[] ca = { 'a', 'e', 'i', 'o', 'u' };
    for(int i = 0; i < ca.length; i++)
        System.out.println("Uppercase: " +
            aa.get(String.valueOf(ca[i])));
}
} ///:~

```

التجسيد الوحيد للصف Dictionary في مكتبة جافا القياسية هو مايسمى بـ *Hashtable*. فهو يمتلك نفس واجهة *AssocArray* لكن فعاليته أكبر، فبدلاً من إجراء البحث الخطي الممل، يستخدم مفتاحاً خاصاً اسمه *hash code* والذي يقوم بأخذ بعض المعلومات المتعلقة بالعنصر المطلوب، وتحويلها إلى قيمة واحدة موافقة لهذا العنصر من نمط *int*. وتمتلك جميع العناصر هذا الترميز، أما الطريقة (*hashCode()*) فهي موجودة في الصف الجذر *Object*.

يوضح البرنامج التالي كيفية استخدام هذا النمط من المجموعات:

```

//: Statistics.java
// Simple demonstration of Hashtable
import java.util.*;
class Counter {
    int i = 1;
    public String toString() {
        return Integer.toString(i);
    }
}
class Statistics {
    public static void main(String[] args) {

```



```

Hashtable ht = new Hashtable();
for(int i = 0; i < 10000; i++) {
    // Produce a number between 0 and 20:
    Integer r =
    new Integer((int) (Math.random() * 20));
    if(ht.containsKey(r))
        ((Counter)ht.get(r)).i++;
    else
        ht.put(r, new Counter());
}
System.out.println(ht);
}
} ///:~

```

أما النتيجة التي ستحصل عليها بعد التنفيذ الأول فهي:

```

{19=526, 18=533, 17=460, 16=513, 15=521,
14=495,
13=512, 12=483, 11=488, 10=487, 9=514, 8=523,
7=497, 6=487, 5=480, 4=489, 3=509, 2=503,
1=475,
0=505}

```

أصبح هنالك مكتبة مجموعات شاملة ...

كما نعلم فإن أهم ما يميز مكتبات لغة C++، مكتبة STL (Standard Template Library) التي تحتوي على العديد من المجموعات المتكاملة، إضافة إلى العديد من الخوارزميات، كالفرز Sorting والبحث Searching التي تعمل مع هذه المجموعات.

لذلك قامت جافا بإنشاء مكتبة عامة مماثلة هي مكتبة JGL (Java Generic Library) والتي تحتوي على العديد من المجموعات، كالقوائم المرتبطة Linked Lists، والمجموعات Sets والأرتال Queues والطباق Maps والمكدسات

Stacks والسلاسل *Sequences* والعدادات المتقدمة، إضافة إلى العديد من خوارزميات الفرز والبحث.

ولقد قامت الشركة المصممة لهذه المكتبة وهي شركة *ObjectSpace* بإتاحة هذه المكتبة بشكل حر لجميع المستخدمين وذلك في الموقع <http://www.ObjectSpace.com>.

ولقد أضافت هذه المكتبة قوة كبيرة للغة جافا وأعطتها العديد من الميزات والإمكانات التي تساعد المبرمج على إنجاز برامجه بشكل مرن وبسيط.

كيفية التعامل مع المجموعات *Collection*؟

يوضح الجدول التالي كل شيء يتعلق بالمجموعات *Collection*، وبالتالي كل ما يمكنك عمله مع المجموعات *Set* والقوائم *List*:

الطريقة	عملها
<code>boolean add(Object)</code>	إضافة العنصر <i>Object</i> إلى المجموعة.
<code>boolean addAll(Collection)</code>	إضافة جميع العناصر المحددة، وتقوم بإرجاع <i>True</i> عند إضافة أي عنصر.
<code>void clear()</code>	لحذف جميع العناصر من المجموعة.
<code>boolean contains(Object)</code>	ترجع <i>True</i> إذا احتوت المجموعة العنصر <i>Object</i> .
<code>boolean isEmpty()</code>	ترجع <i>True</i> إذا كانت المجموعة فارغة.
<code>Iterator iterator()</code>	لإرجاع العداد الذي يمكنك استخدامه للتنقل بين عناصر المجموعة.
<code>boolean remove(Object)</code>	لحذف العنصر <i>Object</i> من المجموعة.
<code>boolean removeAll(Collection)</code>	لحذف جميع العناصر المحددة كوسطاء من المجموعة.
<code>boolean retainAll(Collection)</code>	للاحتفاظ بالعناصر المحددة كوسطاء ضمن



المجموعة فقط.	
لإرجاع عدد عناصر المجموعة.	<code>int size()</code>
لإرجاع مصفوفة تحتوي على جميع عناصر المجموعة <code>Collection</code> .	<code>Object[] toArray()</code>

يوضح المثال التالي كيفية استخدام جميع الطرق السابقة:

```
//: Collection1.java
// Things you can do with all Collections
package c08.newcollections;
import java.util.*;
public class Collection1 {
    // Fill with 'size' elements, start
    // counting at 'start':
    public static Collection
    fill(Collection c, int start, int size) {
        for(int i = start; i < start + size; i++)
            c.add(Integer.toString(i));
        return c;
    }
    // Default to a "start" of 0:
    public static Collection
    fill(Collection c, int size) {
        return fill(c, 0, size);
    }
    // Default to 10 elements:
    public static Collection fill(Collection c) {
        return fill(c, 0, 10);
    }
    // Create & upcast to Collection:
    public static Collection newCollection() {
        return fill(new ArrayList());
        // ArrayList is used for simplicity, but
        // it's
        // only seen as a generic Collection
        // everywhere else in the program.
    }
}
```

```
// Fill a Collection with a range of values:
public static Collection
newCollection(int start, int size) {
    return fill(new ArrayList(), start, size);
}
// Moving through a List with an iterator:
public static void print(Collection c) {
    for(Iterator      x      =      c.iterator();
        x.hasNext();)
        System.out.print(x.next() + " ");
    System.out.println();
}
public static void main(String[] args) {
    Collection c = newCollection();
    c.add("ten");
    c.add("eleven");
    print(c);
    // Find max and min elements; this means
    // different things depending on the way
    // the Comparable interface is implemented:
    System.out.println("Collections.max(c) = "
        +Collections.max(c));
    System.out.println("Collections.min(c) = "
        +Collections.min(c));
    // Add a Collection to another Collection
    c.addAll(newCollection());
    print(c);
    c.remove("3"); // Removes the first one
    print(c);
    c.remove("3"); // Removes the second one
    print(c);
    // Remove all components that are in the
    // argument collection:
    c.removeAll(newCollection());
    print(c);
    c.addAll(newCollection());
    print(c);
    // Is an element in this Collection?
```



```

System.out.println(
    "c.contains(\"4\") = " + c.contains("4"));
// Is a Collection in this Collection?
System.out.println(
    "c.containsAll(newCollection()) = " +
    c.containsAll(newCollection()));
Collection c2 = newCollection(5, 3);
// Keep all the elements that are in both
// c and c2 (an intersection of sets):
c.retainAll(c2);
print(c);
// Throw away all the elements in c that
// also appear in c2:
c.removeAll(c2);
System.out.println("c.isEmpty() = " +
    c.isEmpty());
c = newCollection();
print(c);
c.clear(); // Remove all elements
System.out.println("after c.clear():");
print(c);
}
} ///:~

```

القوائم List؟

يوجد العديد من أنواع القوائم التي تستطيع التعامل معها، كالقوائم العادية List والقوائم المرتبطة بالمصفوفات ArrayList والقوائم المرتبطة .LinkedList يوضح المثال التالي الطرق العديدة التي يمكنك استخدامها مع القوائم :

```

//: List1.java
// Things you can do with Lists
package c08.newcollections;
import java.util.*;
public class List1 {
    // Wrap Collection1.fill() for convenience:
    public static List fill(List a) {

```

```

    return (List)Collection1.fill(a);
}
// You can use an Iterator, just as with a
// Collection, but you can also use random
// access with get():
public static void print(List a) {
    for(int i = 0; i < a.size(); i++)
        System.out.print(a.get(i) + " ");
    System.out.println();
}
static boolean b;
static Object o;
static int i;
static Iterator it;
static ListIterator lit;
public static void basicTest(List a) {
    a.add(1, "x"); // Add at location 1
    a.add("x"); // Add at end
    // Add a collection:
    a.addAll(fill(new ArrayList()));
    // Add a collection starting at location 3:
    a.addAll(3, fill(new ArrayList()));
    b = a.contains("1"); // Is it in there?
    // Is the entire collection in there?
    b = a.containsAll(fill(new ArrayList()));
    // Lists allow random access, which is
    cheap
    // for ArrayList, expensive for LinkedList:
    o = a.get(1); // Get object at location 1
    i = a.indexOf("1"); // Tell index of object
    // indexOf, starting search at location 2:
    i = a.indexOf("1", 2);
    b = a.isEmpty(); // Any elements inside?
    it = a.iterator(); // Ordinary Iterator
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Start at loc 3
    i = a.lastIndexOf("1"); // Last match
    i = a.lastIndexOf("1", 2); // ...after loc2
}

```



```

a.remove(1); // Remove location 1
a.remove("3"); // Remove this object
a.set(1, "y"); // Set location 1 to "y"
// Make an array from the List:
Object[] array = a.toArray();
// Keep everything that's in the argument
// (the intersection of the two sets):
a.retainAll(fill(new ArrayList()));
// Remove elements in this range:
a.removeRange(0, 2);
// Remove everything that's in the
argument:
a.removeAll(fill(new ArrayList()));
i = a.size(); // How big is it?
a.clear(); // Remove all elements
}

public static void iterMotion(List a) {
    ListIterator it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    o = it.next();
    i = it.nextIndex();
    o = it.previous();
    i = it.previousIndex();
}

public static void iterManipulation(List a) {
    ListIterator it = a.listIterator();
    it.add("47");
    // Must move to an element after add():
    it.next();
    // Remove the element that was just
    produced:
    it.remove();
    // Must move to an element after remove():
    it.next();
    // Change the element that was just
    produced:
    it.set("47");
}

```

```

LinkedList ll = new LinkedList();
Collection1.fill(ll, 5);
print(ll);
// Treat it like a stack, pushing:
ll.addFirst("one");
ll.addFirst("two");
print(ll);
// Like "peeking" at the top of a stack:
System.out.println(ll.getFirst());
// Like popping a stack:
System.out.println(ll.removeFirst());
System.out.println(ll.removeFirst());
// Treat it like a queue, pulling elements
// off the tail end:
System.out.println(ll.removeLast());
// With the above operations, it's a
// dequeue!
print(ll);
}

public static void main(String args[]) {
    // Make and fill a new list each time:
    basicTest(fill(new LinkedList()));
    basicTest(fill(new ArrayList()));
    iterMotion(fill(new LinkedList()));
    iterMotion(fill(new ArrayList()));
    iterManipulation(fill(new LinkedList()));
    iterManipulation(fill(new ArrayList()));
    testVisual(fill(new LinkedList()));
    testLinkedList();
}
} ///:~

```

وما الجديد في المجموعات Set؟

لا تختلف المجموعات Set من حيث واجهة الإظهار عن المجموعات الأساسية Collection، سوى أن لها سلوكا مختلفا. فلا تسمح المجموعات Set إلا بوجود ممثل وحيد لكل قيمة عنصر.

ويوجد العديد من أنواع المجموعات كالمجموعات الاعتيادية Set و HashSet و TreeSet و ArraySet.

يوضح البرنامج التالي كيفية التعامل مع المجموعات:

```
//: Set1.java
// Things you can do with Sets
package c08.newcollections;
import java.util.*;
public class Set1 {
    public static void testVisual(Set a) {
        Collection1.fill(a);
        Collection1.fill(a);
        Collection1.fill(a);
        Collection1.print(a); // No duplicates!
        // Add another set to this one:
        a.addAll(a);
        a.add("one");
        a.add("one");
        a.add("one");
        Collection1.print(a);
        // Look something up:
        System.out.println("a.contains(\"one\") : "
            +
            a.contains("one"));
    }
    public static void main(String[] args) {
        testVisual(new HashSet());
        testVisual(new ArraySet());
    }
}
```



} ///:~

نتحدث أخيراً عن الطباق Map...

وهي عبارة عن ثنائيات من المجموعات Collection، أو مانسميها Group of key-value object pairs. لذلك يمكن البحث بسهولة عن قيمة باستخدام مفتاح key.

يوجد العديد من أنواع الطباقات مثل Map و HashMap و TreeMap و ArrayMap.

يوضح هذا البرنامج كيفية التعامل مع هذا النوع من المجموعات:

```
//: Map1.java
// Things you can do with Maps
package c08.newcollections;
import java.util.*;
public class Map1 {
    public final static String[][] testData1 = {
        { "Happy", "Cheerful disposition" },
        { "Sleepy", "Prefers dark, quiet places" },
        { "Grumpy", "Needs to work on attitude" },
        { "Doc", "Fantasizes about advanced
degree"},
        { "Dopey", "'A' for effort" },
        { "Sneezy", "Struggles with allergies" },
        { "Bashful", "Needs self-esteem workshop"},
    };
    public final static String[][] testData2 = {
        { "Belligerent", "Disruptive influence" },
        { "Lazy", "Motivational problems" },
        { "Comatose", "Excellent behavior" }
    };
    public static Map fill(Map m, Object[][] o) {
        for(int i = 0; i < o.length; i++)
            m.put(o[i][0], o[i][1]);
        return m;
    }
}
```

```

}
// Producing a Set of the keys:
public static void printKeys(Map m) {
    System.out.print("Size = " + m.size() + ",
    ");
    System.out.print("Keys: ");
    Collection1.print(m.keySet());
}
// Producing a Collection of the values:
public static void printValues(Map m) {
    System.out.print("Values: ");
    Collection1.print(m.values());
}
// Iterating through Map.Entry objects
(pairs):
public static void print(Map m) {
    Collection entries = m.entries();
    Iterator it = entries.iterator();
    while(it.hasNext()) {
        Map.Entry e = (Map.Entry)it.next();
        System.out.println("Key = " + e.getKey()
        + ", Value = " + e.getValue());
    }
}
public static void test(Map m) {
    fill(m, testData1);
    // Map has 'Set' behavior for keys:
    fill(m, testData1);
    printKeys(m);
    printValues(m);
    print(m);
    String key = testData1[4][0];
    String value = testData1[4][1];
    System.out.println("m.containsKey(\"" + key
    + "\"): " + m.containsKey(key));
    System.out.println("m.get(\"" + key + "\"):
    "+ m.get(key));
    System.out.println("m.containsValue(\""

```



```

+ value + "\\"): " +
m.containsValue(value));
Map m2 = fill(new ArrayMap(), testData2);
m.putAll(m2);
printKeys(m);
m.remove(testData2[0][0]);
printKeys(m);
m.clear();
System.out.println("m.isEmpty(): "
+ m.isEmpty());
fill(m, testData1);
// Operations on the Set change the Map:
m.keySet().removeAll(m.keySet());
System.out.println("m.isEmpty(): "
+ m.isEmpty());
}
public static void main(String args[]) {
    System.out.println("Testing ArrayMap");
    test(new ArrayMap());
    System.out.println("Testing HashMap");
    test(new HashMap());
    System.out.println("Testing TreeMap");
    test(new TreeMap());
}
} ///:~

```

لاحظ بأن الطرق `print()` و `printValues()` و `printKeys()` ليست أدوات مفيدة فقط، لكنها تبين كيفية توليد مشاهد للطبقات `Map` من `Collection`.



تعتمد لغة جافا على فكرة أساسية وهي أن الترميز السيئ لن يعمل بشكل صحيح. وكما في لغة C++ فإن الوقت الأمثل لاكتشاف الأخطاء هو وقت الترجمة *compile time*، وحتى قبل أن تحاول تنفيذ برنامجك. لكن بالطبع لا يمكنك اكتشاف جميع الأخطاء خلال وقت الترجمة، لذلك يمكن معالجة بقية الأخطاء خلال وقت التنفيذ *run time*.

في لغة C وفي العديد من اللغات القديمة، كانت تستخدم العديد من الطرق والأوامر التي تساعد على اكتشاف الأخطاء خلال وقت التنفيذ، لكن هذه الطرق لم تكن فعالة في اكتشاف الأخطاء.

ولقد وُجد أن معالجة الأخطاء باستخدام ما يسمى بالاستثناءات *Exceptions* تساعد كثيراً على اكتشافها خلال وقت التنفيذ. واستخدمت طريقة الاستثناءات هذه في العديد من اللغات، فلغة C++ مثلاً قامت بمعالجة الاستثناءات بالاعتماد على لغة Ada، ولغة جافا اعتمدت على C++ من أجل معالجة الاستثناءات.

الفائدة الأساسية من استخدام الاستثناءات هي أنك في كثير من الأحيان قد لاتعرف ما الذي ستفعله عند حصول مشكلة ما، لكن تعرف تماماً بأنه يتوجب عليك التوقف وعدم المتابعة. وهناك فائدة أخرى أيضاً هي أنها تساعد على معالجة المشكلة في مكان واحد فقط بدلاً من معالجتها في جميع الأماكن التي تتكرر فيها هذه المشكلة، ويتم ذلك باستخدام ما يسمى بمعالج الاستثناء *Exception Handler*. طبعاً هذا يحافظ على ترميزك ويفصل بين الترميز الذي يصف ما الذي تريد عمله عن الترميز الذي يتم تنفيذه في حال حدوث أمر ما أو مشكلة معينة.

في النهاية ستحصل على ترميز أكثر وضوحاً من الترميز القديم.

لنتحدث بتفصيل أكثر عن الاستثناءات...

لكل استثناء شرط *Exception Condition* يمنع استمرار عمل الطريقة أو السرد *scope* (جزء من البرنامج) المنفذ. ومع شرط الاستثناء لا يمكنك متابعة المعالجة لأنك لاتملك المعلومات الضرورية للتعامل مع المشكلة في سياق العمل الحالي *current context*، وكل مايمكنك عمله هو القفز خارج سياق العمل الحالي وإبعاد المشكلة إلى مكان آخر، وهو ما يحدث عندما نقوم بقذف الاستثناء *throw an exception*.



وعندما تقوم بقذف استثناء، تحصل العديد من الأمور: يتم أولاً إنشاء عنصر استثناء *exception object* بنفس الطريقة التي يتم بها إنشاء عنصر جافا وذلك فوق الكومة *heap* باستخدام تعليمة *new*. بعد ذلك يتم إيقاف مسار التنفيذ الحالي ويتم إخراج مؤشر عنصر الاستثناء من سياق العمل الحالي. يقوم بعدها معالج الاستثناء بالبحث عن مكان مناسب لمتابعة تنفيذ البرنامج. هذا المكان هو مانسميه بمعالج الاستثناء *exception handler* والذي يتلخص عمله بمحاولة استرداد المشكلة بحيث يقوم البرنامج إما بتجربة مسار آخر أو يقوم ببساطة بمتابعة مساره الحالي.

كمثال بسيط عما نسميه قذف استثناء *throwing an exception*، لنفترض لدينا مؤشر عنصر بالاسم *t* (من الممكن أن يتم تمرير مؤشر لم يتم تحديد قيمته الابتدائية) ولنفترض بأنك ترغب بالتحقق من عدم إعطائه قيمة ابتدائية قبل أن تحاول استدعاء طريقة تستخدم مؤشر العنصر هذا. تستطيع إرسال المعلومات حول الخطأ بإنشاء عنصر يمثل هذه المعلومات ومن ثم قذف العنصر خارج سياق العمل الحالي، وذلك على الشكل:

```
if(t == null)
    throw new NullPointerException();
```

ويمكن للاستثناء أن يمتلك وسطاء *arguments*، فمثلاً يمكننا كتابة مايلي:

```
if(t == null)
    throw new NullPointerException("t = null");
```

كيف يتم إذاً التقاط استثناء...

عندما تقوم بطريقة ما بقذف استثناء، يجب أن تأخذ على عاتقها قضية التقاط هذا الاستثناء والتعامل معه. ومن أهم مميزات عملية معالجة الاستثناءات في لغة جافا هي أنها تسمح لك بالتركيز على المشكلة التي تحاول حلها في مكان ما، ثم معالجة الأخطاء الناتجة عن هذا الترميز في مكان آخر.

لرؤية كيفية التقاط استثناء يتوجب عليك أولاً التعرف على مفهوم المنطقة المحمية *guarded region* والتي هي عبارة عن جزء من الترميز الذي قد يولد استثناءات، متبوعاً بالترميز الذي يعالج هذه الاستثناءات.

يمكنك التقاط استثناء باستخدام كتلة *try...*

عندما نقوم بقذف استثناء من داخل طريقة، فإنه سيتم الخروج من هذه الطريقة أثناء عملية قذف الاستثناء. لكن في حال أردت عدم الخروج من الطريقة يمكنك إنشاء كتلة *block* خاصة تمكن هذه الطريقة من التقاط الاستثناء. تسمى هذه الكتلة بكتلة المحاولة *try block* لأنك تجرب مختلف طرق استدعاء الطريقة ضمنها. وهي تأخذ الشكل:

```
try {
    // Code that might generate exceptions
}
```

فإذا افترضنا أنك تقوم بالتحقق من الأخطاء الموجودة في لغة برمجة لاتدعم معالجة الاستثناءات، ستضطر للقيام بالتحقق من كل طريقة واختبار أخطاء الترميز، حتى لو قمت باستدعاء نفس الطريقة مرات عديدة.

أما باستخدام معالج الاستثناء *Exception Handler*، يتم وضع كل شيء في كتلة المحاولة والنقاط جميع الاستثناءات في مكان واحد. هذا يعني بأن ترميزك سيصبح أكثر سهولة كتابة وقراءة لأن هدف الترميز سوف لن يتضارب مع التحقق من الأخطاء. وتنتهي عملية قذف الاستثناء في مكان هو مؤشر الاستثناء *exception handler*، حيث يوجد مؤشر واحد لكل نمط استثناء ترغب بالنقاطه، ويأتي دائماً بعد كتلة المحاولة *try block* ويأخذ الشكل:

```
try {
    // Code that might generate exceptions
} catch (Type1 id1) {
    // Handle exceptions of Type1
} catch (Type2 id2) {
    // Handle exceptions of Type2
} catch (Type3 id3) {
    // Handle exceptions of Type3
}
```




```
}
// etc...
```

وكل عبارة التقاط *catch clause* (أو مؤشر استثناء) تشبه طريقة بسيطة تأخذ بسيطاً وحيداً فقط من نمط خاص. ويجب أن تظهر المؤشرات مباشرة بعد كتلة المحاولة *try block*. فإذا تم قذف استثناء يقوم معالج الاستثناء باصطياد أول مؤشر يتوافق وسيطه مع نمط الاستثناء. يقوم بعدها بالدخول إلى عبارة الالتقاط *catch clause* ومعالجة هذا الاستثناء.

هناك طريقة إجبارية لتوصيف استثناء...

كي تستطيع إعلام المبرمج الزبون *client programmer* بالاستثناءات التي يمكن لطريقة ما قذفها، اكتب كلمة المفتاح *throws* بعد اسم الطريقة ثم اتبعها بقائمة أنماط الاستثناءات الخاصة بهذه الطريقة كما في المثال التالي:

```
void f() throws tooBig, tooSmall, divZero {
//...
```

أما لو كتبت:

```
void f() { // ...
```

فهذا يعني بأنه لا يمكن لهذه الطريقة قذف أي استثناء (عدا الاستثناء ذو النمط *RuntimeException* الذي يمكن قذفه من أي مكان كما سنرى لاحقاً).

انتبه فلا يمكنك أن تكذب أبداً عند توصيف الاستثناء، فإذا سببت طريقة ما استثناءات لم تقم بمعالجتها، سيكتشف المترجم ذلك وسيطلب منك إما معالجة هذه الاستثناءات أو تحديدها بشكل دقيق ضمن جزء توصيف الاستثناءات *exception specification*. أما الشيء الوحيد الذي يمكنك الكذب فيه فهو الادعاء بقذف استثناء. سيأخذ المترجم عندها اسم هذا الاستثناء ويجبر مستخدمك بطريقة بمعالجة هذا الاستثناء كما لو أنه موجود فعلاً.

وكيف أستطيع التقاط أي استثناء؟

تستطيع إنشاء مؤشر يقوم بالتقاط أي نمط استثناء وذلك عن طريق التقاط نمط الاستثناء الأساسي *Exception* على الشكل:

```
catch(Exception e) {
    System.out.println("caught an exception");
}
```

أما الطرق التي يمكنك استخدامها مع نمط الاستثناء الأساسي *Exception* فهي:

- ✓ *String getMessage()* : لقراءة الرسالة المفصلة.
- ✓ *String toString()* : لإرجاع وصف مبسط عن النمط الأساسي *Throwable* إضافة إلى الرسالة المفصلة في حال وجودها.
- ✓ *void printStackTrace()* : لطباعة عنصر *Throwable* وكذلك أثر مكدس استدعاء عنصر *Throwable* على الخطأ القياسي *Standard error*.
- ✓ *void printStackTrace(PrintStream)* : لطباعة عنصر *Throwable* وكذلك أثر مكدس استدعاء عنصر *Throwable* على سلسلة من اختياريك.

يوضح المثال التالي استخدام طرق الصف *Exception*:

```
//: ExceptionMethods.java
// Demonstrating the Exception Methods
package c09;
public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("Here's my
                Exception");
        } catch(Exception e) {
            System.out.println("Caught Exception");
            System.out.println(
                "e.getMessage(): " + e.getMessage());
        }
    }
}
```



```

System.out.println(
    "e.toString(): " + e.toString());
System.out.println("e.printStackTrace():");
e.printStackTrace();
}
}
} ///:~

```

أما خرج هذا البرنامج فسيكون على الشكل:

```

Caught Exception
e.getMessage(): Here's my Exception
e.toString(): java.lang.Exception: Here's my
Exception
e.printStackTrace():
java.lang.Exception: Here's my Exception
    at ExceptionMethods.main

```

وقد نحتاج في بعض الأحيان لإعادة قذف استثناء قمت بالتقاطه من قبل، خاصةً عند تستخدم *Exception* لالتقاط أي استثناء. في هذه الحالة قم فقط بإعادة قذف مؤشر الاستثناء كما في الشكل:

```

catch(Exception e) {
    System.out.println("An exception was
    thrown");
    throw e;
}

```

ماهي الاستثناءات القياسية في لغة جافا؟

تحتوي لغة جافا على الصف *Throwable* الذي يقوم بتوصيف أي شيء يمكنك قذفه كاستثناء. ويوجد نمطان لعناصر هذا الصف: النمط الأول هو *Error* ويمثل أخطاء النظام وأخطاء الترجمة التي لن تقلق أبداً بالنسبة لعملية التقاطها، أما النمط الثاني فهو *Exception*، وهو النمط الأساسي الذي يمكنك قذفه من خلال أي طريقة من طرق

صفوف مكتبة جافا القياسية، أو من خلال الطرق التي تقوم بإنشائها أو من خلال الحوادث التي تحصل وقت التنفيذ.

توجد العديد من الاستثناءات القياسية، لكن يجب عليك أولاً فهم الصف `java.lang.Exception` الذي يعتبر صف الاستثناء الأساسي الذي يمكن لبرنامجك التقاطه. أما بقية الاستثناءات فهي مشتقة من هذا الاستثناء.

توجد أيضاً مجموعة من الاستثناءات التي يتم قذفها تلقائياً من قبل جافا ولن تكون بحاجة أبداً لتضمينها ضمن توصيف الاستثناء، هذه المجموعة موجودة في الصف `.RuntimeException`.

يوضح المثال السابق كيفية استخدام الاستثناءات القياسية.

يمكنك أيضاً إنشاء استثناءاتك الخاصة...

بالطبع لن تجبرك جافا على استخدام استثناءاتها، فهي تعلم بأنك تحتاج في كثير من الأحيان لإنشاء استثناءات خاصة بك لمعالجة بعض الأخطاء الخاصة. الشيء الذي تجبرك جافا على القيام به هو توريث استثناءك من نمط استثناء موجود من قبل كما في المثال التالي:

```
//: Inheriting.java
// Inheriting your own exceptions
class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}
public class Inheriting {
    public static void f() throws MyException {
        System.out.println(
            "Throwing MyException from f()");
        throw new MyException();
    }
}
```



```

public static void g() throws MyException {
    System.out.println(
        "Throwing MyException from g()");
    throw new MyException("Originated in g()");
}
public static void main(String[] args) {
    try {
        f();
    } catch(MyException e) {
        e.printStackTrace();
    }
    try {
        g();
    } catch(MyException e) {
        e.printStackTrace();
    }
}
} ///:~

```

ويتم التوريث عند إنشاء الصف الجديد:

```

class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}

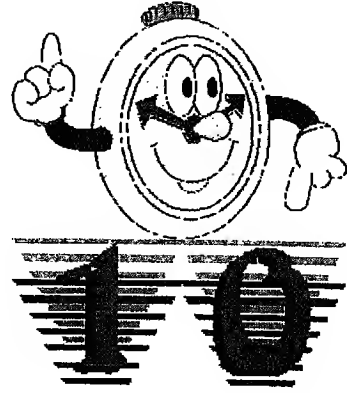
```

أما خرج هذا البرنامج فسيكون على الشكل:

```

Throwing MyException from f()
MyException
    at Inheriting.f(Inheriting.java:16)
    at Inheriting.main(Inheriting.java:24)
Throwing MyException from g()
MyException: Originated in g()
    at Inheriting.g(Inheriting.java:20)
    at Inheriting.main(Inheriting.java:29)

```

عملية إنشاء نظام إدخال وإخراج *input/output* جيد من أصعب

المهام التي تواجه مصممي لغات البرمجة.

تعتبر

ولقد حلت جافا هذه المشكلة بإنشاء الكثير من الصفوف الخاصة بنظام

الإدخال والإخراج.

ووفقاً لخاصية التوريث، جميع الصفوف المشتقة من الصف الأساسي *Reader* تمتلك طريقة أساسية *read()* لقراءة بايت وحيد أو مصفوفة بايتات. كذلك تمتلك جميع الصفوف المشتقة من الصف الأساسي *write* طريقة أساسية *write()* لكتابة بايت وحيد أو مصفوفة من البايتات. (جميع الصفوف التي سنتحدث عنها في هذا الفصل معروفة في الإصدار *Java 1.1*).

أنماط الصف *Reader*...

إن الهدف من الصف *Reader* هو تمثيل الصفوف التي تقوم بتوليد الإدخالات من مصادر مختلفة، هذه المصادر هي:

١. مصفوفة بايتات.
 ٢. عنصر من نمط *String*.
 ٣. ملف *file*.
 ٤. أنبوب *pipe*.
 ٥. سلاسل دفق *stream*، يمكن تجميعها سوية لتوليد دفق وحيد.
 ٦. مصادر أخرى مثل اتصال إنترنت *Internet Connection*.
- يوضح الجدول التالي أنماط الصف *Reader* وعمل كل منها:

الصف	عمله
<i>CharArrayReader</i>	السماح باستخدام دارئ <i>buffer</i> ذاكرة كعنصر <i>Reader</i> .
<i>StringReader</i>	لتحويل عنصر <i>String</i> إلى <i>Reader</i> .
<i>FileReader</i>	للقراءة من ملف.
<i>PipedReader</i>	لتوليد المعلومات التي تمت كتابتها في <i>PipedWriter</i> الموافق.
<i>FilterReader</i>	عبارة عن صف مجرد يمثل واجهة للديكورات

بأدوات هامة لصفوف Reader الأخرى.	<i>decorators interface</i> التي تزودك
----------------------------------	--

أما الصف *FliterReader* فيملك مجموعة من الأنماط موضحة في الجدول التالي:

الصف	عمله
<i>DataInputStream</i>	لقراءة العناصر الأولية (<i>int, char, long, ...</i>) من دفق <i>stream</i> .
<i>BufferedReader</i>	يفيد في منع القراءة الفيزيائية في كل مرة نحتاج فيها إلى قراءة معطيات إضافية.
<i>LineNumberReader</i>	للاحتفاظ بأرقام الأسطر في دفق الدخل <i>input stream</i> .
<i>PushbackReader</i>	يستخدم لدفع آخر محرف مقروء إلى الوراء، وعادةً يستخدمه المترجم.

أنماط الصف *Writer*...

يوضح الجدول التالي أنماط الصف *Writer* وعمل كل منها:

الصف	عمله
<i>CharArrayWriter</i>	لإنشاء دارئ <i>buffer</i> في الذاكرة، حيث تخزن فيه جميع المعطيات التي تقوم بإرسالها إلى دفق <i>stream</i> .
<i>StringWriter</i>	لتحويل عنصر <i>String</i> إلى <i>Writer</i> .
<i>FileWriter</i>	للكتاب في ملف.
<i>PipedWriter</i>	يتم تحويل جميع المعلومات التي تكتب في هذا الصف تلقائياً إلى <i>PipedReader</i> الموافق.

عبارة عن صف مجرد يمثل واجهة للديكورات decorators interface التي تزودك بأدوات هامة لصفوف Writer الأخرى.	FilterWriter
--	--------------

أما الصف FliterWriter فيملك مجموعة من الأنماط الموضحة في الجدول التالي:

الصف	عمله
DataOutputStream	كتابة العناصر الأولية (int, char, long, ...) في دفق stream.
BufferedWriter	يفيد في منع الكتابة الفيزيائية في كل مرة نحتاج فيها إلى كتابة معطيات إضافية.
PrintWriter	لتوليد خرج منسق formatted output.

وعلى الرغم من وجود الكثير من صفوف الدخل والخرج والتي يمكن دمجها بعدة طرق، إلا أن عليك الانتباه من أجل إجراء عمليات الدمج بشكل صحيح. يوضح المثال التالي كيفية إنشاء واستخدام أنماط الدخل والخرج التقليدية، ويمكنك اعتماده كمرجع عند قيامك بكتابة ترميزك الخاصة:

```
//: NewIODemo.java
// Java 1.1 IO typical usage
import java.io.*;
public class NewIODemo {
    public static void main(String[] args) {
        try {
            // 1. Reading input by lines:
            BufferedReader in =
                new BufferedReader(
                    new FileReader(args[0]));
            String s, s2 = new String();
            while((s = in.readLine()) != null)
                s2 += s + "\n";
            in.close();
            // 1b. Reading standard input:
```

```

BufferedReader stdin =
    new BufferedReader(
        new InputStreamReader(System.in));
System.out.print("Enter a line:");
System.out.println(stdin.readLine());
// 2. Input from memory
StringReader in2 =
    new StringReader(s2);
int c;
while((c = in2.read()) != -1)
    System.out.print((char)c);
// 3. Formatted memory input
try {
    DataInputStream in3 =
        new DataInputStream(
            // Oops: must use deprecated class:
            new StringBufferInputStream(s2));
    while(true)
        System.out.print((char)in3.readByte(
        ));
} catch(EOFException e) {
    System.out.println("End of stream");
}
// 4. Line numbering & file output
try {
    LineNumberReader li =
        new LineNumberReader(
            new StringReader(s2));
    BufferedReader in4 =
        new BufferedReader(li);
    PrintWriter out1 =
        new PrintWriter(
            new BufferedWriter(
                new FileWriter("IODemo.out")));
    while((s = in4.readLine()) != null )
        out1.println(
            "Line " + li.getLineNumber() + s);
    out1.close();
}

```

```

    } catch (EOFException e) {
        System.out.println("End of stream");
    }
    // 5. Storing & recovering data
    try {
        DataOutputStream out2 =
            new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("Data.txt")));
        out2.writeDouble(3.14159);
        out2.writeBytes("That was pi");
        out2.close();
        DataInputStream in5 =
            new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream("Data.txt")));
        BufferedReader in5br =
            new BufferedReader(
                new InputStreamReader(in5));
        // Must use DataInputStream for
        data:
        System.out.println(in5.readDouble());
        ;
        // Can now use the "proper"
        readLine():
        System.out.println(in5br.readLine());
        ;
    } catch (EOFException e) {
        System.out.println("End of stream");
    }
    } catch (FileNotFoundException e) {
        System.out.println(
            "File Not Found:" + args[1]);
    } catch (IOException e) {
        System.out.println("IO Exception");
    }
}
} ///:~

```

في الفقرة 1 من البرنامج السابق نقوم بقراءة سطر دخل فقط من أجل تغليف عنصر `BufferedReader` حول عنصر `FileReader`. أما الفقرة 1b فتوضح كيفية استخدام الدخل القياسي `System.in` من أجل القراءة من محكم `Console` الدخل. أما في الفقرة 2 فتلاحظ بأنه إذا كان لديك عنصر `String` وأردت القراءة منه، فإمكانك استخدام عنصر من نمط `StringReader`.

وفي الفقرة 3، وضعنا مشكلة صغيرة في تصميم مكتبة دفق الدخل والخرج، حيث تظهر رسالة تطلب عدم استخدام الطريقة `StringBufferInputStream` لأنها تعتبر من طرق مكتبة جافا القديمة، وتتصحك باستخدام الطريقة `StringReader`. ولتستطيع القيام بإجراء عملية دخل منسقة `formatted memory input` عليك استخدام الباني `DataInputStream` الذي يحتاج بدوره إلى الوسيط `InputStream`.

بينما في الفقرة 4 ستجد أنك مجبر على استخدام الصيغتين القديمتين `DataInputStream` و `DataOutputStream` بسبب عدم وجود أي مقليل لهما في المكتبة الجديدة. أخيراً في الفقرة 5 ستجد أن عملية فرز واسترداد المعطيات أصبحت أكثر سهولة باستخدام المكتبات السابقة.

ماهي الفائدة من استخدام الصف `File`؟

للوهلة الأولى قد يخيّل إليك بأن هذا الصف يدلّ على ملفّ، لكن الأمر ليس كذلك. فالصف `File` قد يمثّل اسم ملفّ خاصّ، أو أسماء مجموعة ملفات موجودة في مجلّد و في هذه الحالة يمكنك استخدام الطريقة `list()` لمعرفة تلك الملفات. البرنامج التالي يوضح كيفية استخدام الطريقة `list()` لإظهار جميع الملفات الموجودة ضمن مجلّد، أو لإظهار قائمة مقيدة من الملفات وذلك باستخدام مرشّح المجلّد `directory filter`.

```

//: DirList.java
// Displays directory listing
package cl0;
import java.io.*;
public class DirList {
    public static void main(String[] args) {
        try {
            File path = new File(".");
            String[] list;
            if(args.length == 0)
                list = path.list();
            else
                list = path.list(new DirFilter(args[0]));
            for(int i = 0; i < list.length; i++)
                System.out.println(list[i]);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

class DirFilter implements FilenameFilter {
    String afn;
    DirFilter(String afn) { this.afn = afn; }
    public boolean accept(File dir, String
        name) {
        // Strip path information:
        String f = new File(name).getName();
        return f.indexOf(afn) != -1;
    }
}
} ///:~

```

لاحظ بأن الصف `DirFilter` ينفذ الواجهة `FilenameFilter` التي تم شرحها في الفصل السابع والتي تأخذ الشكل:

```

public interface FilenameFilter {
    boolean accept(File dir, String name);
}

```

وهذه الواجهة تروّنا بالطريقة `accept()` والتي تساعد الطريقة `list()` على تحديد أسماء الملفات المتضمنة في القائمة. وتأخذ الطريقة `accept()` وسيطين الأول من نمط `File` ويمثل اسم المجلّد الذي يحتوي الملف، أما الثاني فهو عبارة عن عنصر `String` يتضمّن اسم هذا الملف.

لاحظ أيضاً كيف يتم استخدام الطريقة `getName()` للحصول على اسم الملف، ثم يتم بعد ذلك البحث عنه باستخدام الطريقة `indexOf()` والتي ترجع القيمة -1 في حال لم يتم العثور عليه.

ولا يقتصر دور الصف `File` على تمثيل الطريق لمجلّد موجود مسبقاً، أو تمثيل ملف أو عدة ملفات، بل يمكن استخدام عنصر `File` لإنشاء مجلد جديد أو طريق مجلد كامل إذا لم يكن موجوداً من قبل. يمكنه أيضاً المساعدة في الحصول على خصائص الملفات (أحجامها أو تاريخ آخر تعديل ... الخ)، كذلك يمكنه معرفة فيما إذا كان العنصر `File` يمثل ملفاً أو مجلداً، وحتى القيام بحذف ملف.

يوضح البرنامج التالي كيفية القيام بالعمليات السابقة:

```
//: MakeDirectories.java
// Demonstrates the use of the File class to
// create directories and manipulate files.
import java.io.*;
public class MakeDirectories {
    private final static String usage =
        "Usage:MakeDirectories path1 ...\n" +
        "Creates each path\n" +
        "Usage:MakeDirectories -d path1 ...\n" +
        "Deletes each path\n" +
        "Usage:MakeDirectories -r path1 path2\n" +
        "Renames from path1 to path2\n";
    private static void usage() {
        System.err.println(usage);
        System.exit(1);
    }
    private static void fileData(File f) {
        System.out.println(
            "Absolute path: " + f.getAbsolutePath() +
```

```

"\n Can read: " + f.canRead() +
"\n Can write: " + f.canWrite() +
"\n getName: " + f.getName() +
"\n getParent: " + f.getParent() +
"\n getPath: " + f.getPath() +
"\n length: " + f.length() +
"\n lastModified: " + f.lastModified());
if(f.isFile())
    System.out.println("it's a file");
else if(f.isDirectory())
    System.out.println("it's a directory");
}
public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
        if(args.length != 3) usage();
        File
            old = new File(args[1]),
            rname = new File(args[2]);
        old.renameTo(rname);
        fileData(old);
        fileData(rname);
        return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
        count++;
        del = true;
    }
    for( ; count < args.length; count++) {
        File f = new File(args[count]);
        if(f.exists()) {
            System.out.println(f + " exists");
            if(del) {
                System.out.println("deleting..." + f);
                f.delete();
            }
        }
    }
}

```



```

    }
    else { // Doesn't exist
        if(!del) {
            f.mkdirs();
            System.out.println("created " + f);
        }
    }
    fileData(f);
}
}
} ///:~

```

ضمن الطريقة `fileData()` يمكنك رؤية كيفية استخدام الطرق المختلفة لإظهار المعلومات حول الملف أو طريق المجلد. أما الطريقة `renameTo()` المستخدمة ضمن البرنامج الرئيسي `main()`، فتستخدم لتغيير اسم ملف إلى طريق جديد كامل يعبر عنه من خلال الوسيط.

بإمكانك تقسيم نص باستخدام الصف *Stream* ...*Tokenizer*

لا يعمل هذا الصف إلا مع عناصر `Reader` على الرغم من أنه غير مشتق من الصف `Reader` أو الصف `Writer`. ويستخدم هذا الصف لتقسيم أي عنصر `Reader` إلى سلسلة من العلامات `tokens`. وهذه العلامات عبارة عن بنات نصوص تفصل بينها فواصل حسب اختيارك. فمثلاً يمكن أن تكون العلامات عبارة عن كلمات `words` تفصل بينها فراغات وعلامات تنقيط. يوضح البرنامج التالي كيفية القيام بتعداد تكرار الكلمات في ملف نصي:

```

//: SortedWordCount.java
// Counts words in a file, outputs
// results in sorted form.
import java.io.*;

```

سلسلة الرضا للمعلومات

```

import java.util.*;
import c08.*; // Contains StrSortVector
class Counter {
    private int i = 1;
    int read() { return i; }
    void increment() { i++; }
}
public class SortedWordCount {
    private FileReader file;
    private StreamTokenizer st;
    private Hashtable counts = new Hashtable();
    SortedWordCount(String filename)
        throws FileNotFoundException {
        try {
            file = new FileReader(filename);
            st = new StreamTokenizer(file);
            st.ordinaryChar('.');
            st.ordinaryChar('-');
        } catch (FileNotFoundException e) {
            System.out.println(
                "Could not open " + filename);
            throw e;
        }
    }
    void cleanup() {
        try {
            file.close();
        } catch (IOException e) {
            System.out.println(
                "file.close() unsuccessful");
        }
    }
    void countWords() {
        try {
            while(st.nextToken() !=
                StreamTokenizer.TT_EOF) {
                String s;
                switch(st.ttype) {

```

```

        case StreamTokenizer.TT_EOL:
            s = new String("EOL");
            break;
        case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
        case StreamTokenizer.TT_WORD:
            s = st.sval; // Already a String
            break;
        default: // single character in ttype
            s = String.valueOf((char)st.ttype);
    }
    if(counts.containsKey(s))
        ((Counter)counts.get(s)).increment();
    else
        counts.put(s, new Counter());
    }
} catch(IOException e) {
    System.out.println(
        "st.nextToken() unsuccessful");
}
}

Enumeration values() {
    return counts.elements();
}

Enumeration keys() { return counts.keys(); }
Counter getCounter(String s) {
    return (Counter)counts.get(s);
}

Enumeration sortedKeys() {
    Enumeration e = counts.keys();
    StrSortVector sv = new StrSortVector();
    while(e.hasMoreElements())
        sv.addElement((String)e.nextElement());
    // This call forces a sort:
    return sv.elements();
}

public static void main(String[] args) {

```

```

try {
    SortedWordCount wc =
        new SortedWordCount(args[0]);
    wc.countWords();
    Enumeration keys = wc.sortedKeys();
    while(keys.hasMoreElements()) {
        String key = (String)keys.nextElement();
        System.out.println(key + ": "
            + wc.getCounter(key).read());
    }
    wc.cleanup();
} catch(Exception e) {
    e.printStackTrace();
}
}
} ///:~

```

كما تلاحظ من البرنامج السابق، فمن أجل فتح ملف تم استخدام الصف *FileReader*. أما من أجل تحويل ملف إلى مجموعة كلمات فتم إنشاء عنصر *StreamTokenizer* اعتماداً على الصف *FileReader*. ويحتوي الصف *StreamTokenizer* على قائمة افتراضية من الفواصل (ويمكنك إضافة فواصل أخرى إليها باستخدام طرق عديدة). فمثلاً تستخدم الطريقة *ordinaryChar()* لإهمال أية أحرف وعدم تضمينها في قائمة الكلمات التي قمت بإنشائها. فإذا كتبنا *st.ordinaryChar('.')* فهذا يعني بأنه لن يتم تضمين النقطة كجزء من قائمة الكلمات.

أما في الطريقة *countWords()* فيتم جرّ العلامات *tokens* واحدة تلو الأخرى، وتستخدم معلومات *ttype* لتحديد ما سيتم عمله مع كل علامة، لأن العلامة قد تكون نهاية سطر أو قد تكون رقماً أو سلسلة أو حرفاً وحيداً. وبعد إيجاد العلامة، يتم الاستعلام في عدادات *Hashtable* لمعرفة إن كانت تحتوي على هذه العلامة كمفتاح *key* أم لا. ففي حال وجود هذه العلامة تتم زيادة العنصر *Counter* التي تؤكد بأنه تم إيجاد ممثلاً آخر عن هذه الكلمة. أما في حال عدم وجود هذه العلامة فيتم إنشاء عداد *Counter* جديداً.

يمكنك القيام بنفس العمل باستخدام الصف

...StringTokenizer

يتشابه الصفان *StringTokenizer* و *StreamTokenizer* إلى حد كبير من حيث عملهما.

ويقوم الصف *StringTokenizer* بإرجاع العلامات ضمن سلسلة واحدة تلو الأخرى. هذه العلامات عبارة عن محارف متتابعة مفصولة إما بمحارف *tab* أو بفراغات أو بمحارف أسطر جديدة *newlines*. لذلك فإنّ علامات السلسلة "Where is my cat?" هي "Where" و "is" و "my" و "cat".

وتماماً كما في *StreamTokenizer* يمكنك إخبار *StringTokenizer* بتقسيم الدخل بالطريقة التي تريدها، لكن مع *StringTokenizer* فإنّك تقوم بتمرير وسيط ثانٍ إلى الباني الذي يمثل الفاصل الذي ترغب باستخدامه ضمن عنصر *.String*.

والمثال التالي يوضح كيفية استخدام هذا الصف:

```
//: AnalyzeSentence.java
// Look for particular sequences
// within sentences.
import java.util.*;
public class AnalyzeSentence {
    public static void main(String[] args) {
        analyze("I am happy about this");
        analyze("I am not happy about this");
        analyze("I am not! I am happy");
        analyze("I am sad about this");
        analyze("I am not sad about this");
        analyze("I am not! I am sad");
        analyze("Are you happy about this?");
        analyze("Are you sad about this?");
        analyze("It's you! I am happy");
        analyze("It's you! I am sad");
    }
}
```

```

}
static StringTokenizer st;
static void analyze(String s) {
    prt("\nnew sentence >> " + s);
    boolean sad = false;
    st = new StringTokenizer(s);
    while (st.hasMoreTokens()) {
        String token = next();
        // Look until you find one of the
        // two starting tokens:
        if(!token.equals("I") &&
            !token.equals("Are"))
            continue; // Top of while loop
        if(token.equals("I")) {
            String tk2 = next();
            if(!tk2.equals("am")) // Must be after I
                break; // Out of while loop
            else {
                String tk3 = next();
                if(tk3.equals("sad")) {
                    sad = true;
                    break; // Out of while loop
                }
            }
            if (tk3.equals("not")) {
                String tk4 = next();
                if(tk4.equals("sad"))
                    break; // Leave sad false
                if(tk4.equals("happy")) {
                    sad = true;
                    break;
                }
            }
        }
    }
    if(token.equals("Are")) {
        String tk2 = next();
        if(!tk2.equals("you"))
            break; // Must be after Are
    }
}

```

```

String tk3 = next();
if(tk3.equals("sad"))
    sad = true;
break; // Out of while loop
}
}
if(sad) prt("Sad detected");
}
static String next() {
    if(st.hasMoreTokens()) {
        String s = st.nextToken();
        prt(s);
        return s;
    }
    else
        return "";
}
static void prt(String s) {
    System.out.println(s);
}
} ///:~

```

إعادة توجيه الدخل والخرج القياسي...

ابتداءً من النسخة 1.1 java تمت إضافة مجموعة من الطرق إلى الصف *System* من أجل إعادة توجيه الدخل والخرج القياسي، وهذه الطرق هي:

setIn(InputStream)

setOut(PrintStream)

setErr(PrintStream)

وتفيد عملية إعادة توجيه الخرج خاصةً عندما تقوم بشكل مفاجئ بتوليد كمية كبيرة من الخرج على شاشتك بحيث تنزلق بسرعة دون أن تتمكن من قراءتها بشكل جيد.

أما إعادة توجيه الدخل فهي مفيدة من أجل برامج سطر الأوامر *command-line* *program* التي تحتاج فيها إلى اختبار سلسلة دخل مستخدم *user-input* خاصة بشكل متكرر.

والمثال البسيط التالي يوضح كيفية استخدام هذه الطرق:

```
//: Redirecting.java
// Demonstrates the use of redirection for
// standard IO in Java 1.1
import java.io.*;
class Redirecting {
    public static void main(String[] args) {
        try {
            BufferedInputStream in =
                new BufferedInputStream(
                    new FileInputStream(
                        "Redirecting.java"));
            // Produces deprecation message:
            PrintStream out =
                new PrintStream(
                    new BufferedOutputStream(
                        new FileOutputStream("test.out")));
            System.setIn(in);
            System.setOut(out);
            System.setErr(out);
            BufferedReader br =
                new BufferedReader(
                    new InputStreamReader(System.in));
            String s;
            while((s = br.readLine()) != null)
                System.out.println(s);
            out.close(); // Remember this!
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
} ////:~
```


يقوم هذا البرنامج بربط الدخل القياسي في ملف، كما يقوم بإعادة توجيه الخرج القياسي والخطأ القياسي في ملف آخر.

بإمكانك ضغط بياناتك أيضاً !!؟

ابتداءً من النسخة 1.1 Java تمت إضافة بعض الصفوف التي تدعم عملية قراءة وكتابة الدفق *stream* بطريقة مضغوطة. لكن انتبه فالصفوف السابقة غير مشتقة من الصفتين *Reader* و *Writer* بل من الصفتين القديمتين *InputStream* و *OutputStream*. والجدول التالي يبين صفوف الضغط وعمل كل منها:

العمل	اسم صف الضغط
تقوم الطريقة <i>GetChecksum()</i> بالتحقق من مجموع أي عنصر <i>InputStream</i> بالإضافة إلى فك الضغط.	<i>CheckedInputStream</i>
تقوم الطريقة <i>GetChecksum()</i> بالتحقق من مجموع أي عنصر <i>OutputStream</i> بالإضافة إلى فك الضغط.	<i>CheckedOutputStream</i>
صف أساسي لجميع صفوف الضغط.	<i>DeflaterOutputStream</i>
مشتق من <i>DeflaterOutputStream</i> يفيد في ضغط المعطيات بنمط ملف <i>.Zip</i> .	<i>ZipOutputStream</i>
مشتق من <i>DeflaterOutputStream</i> يفيد في ضغط المعطيات بنمط ملف <i>.GZip</i> .	<i>GZipOutputStream</i>
صف أساسي لجميع صفوف فك الضغط.	<i>InflaterInputStream</i>
مشتق من <i>DeflaterInputStream</i> يفيد في فك ضغط المعطيات التي تم تخزينها	<i>ZipInputStream</i>

بنمط ملف Zip.	
مشتق من DeflaterOutputStream	GZipInputStream
يفيد في فك ضغط المعطيات التي تم تخزينها بنمط ملف GZip.	

وعلى الرغم من وجود الكثير من خوارزميات الضغط، إلا أن الضغط بنمط Zip أو Gzip أكثر استخداماً. لذلك يمكنك ببساطة التعامل مع المعطيات المضغوطة باستخدام الكثير من الأدوات المتاحة لقراءة وكتابة هذين النمطين. وتعتبر واجهة Gzip الأكثر بساطة، لذلك فهي الأفضل استخداماً عندما تحتاج إلى ضغط معطيات دفق وحيد.

يوضح المثال التالي كيفية ضغط ملف وحيد:

```
//: GZIPcompress.java
// Uses Java 1.1 GZIP compression to compress
// a file whose name is passed on the command
// line.
import java.io.*;
import java.util.zip.*;
public class GZIPcompress {
    public static void main(String[] args) {
        try {
            BufferedReader in =
                new BufferedReader(
                    new FileReader(args[0]));
            BufferedOutputStream out =
                new BufferedOutputStream(
                    new GZIPOutputStream(
                        new FileOutputStream("test.gz")));
            System.out.println("Writing file");
            int c;
            while((c = in.read()) != -1)
                out.write(c);
            in.close();
            out.close();
            System.out.println("Reading file");
        }
    }
}
```

```
BufferedReader in2 =  
    new BufferedReader(  
        new InputStreamReader(  
            new GZIPInputStream(  
                new FileInputStream("test.gz"))));  
String s;  
while((s = in2.readLine()) != null)  
    System.out.println(s);  
} catch(Exception e) {  
    e.printStackTrace();  
}  
}  
} ///:~
```

هناك أيضاً أداة ممتازة للأرشفة...

يمكن استخدام أنماط ملفات الأرشفة *JAR* (*Java Archive*) لجمع عدة ملفات في ملف مضغوط وحيد بنمط *Zip* مثلاً. ويمكن استخدام ملفات *JAR* في أي منصة عمل *platform* (مثل أي شيء في جافا)، يمكنك أيضاً تضمين ملفات صوت وصورة إضافة إلى ملفات صف.

وتعتبر ملفات *JAR* مفيدة جداً خاصة عندما تتعامل مع الإنترنت. فسابقاً كان يتوجب على مستعرض *Web* إرسال طلبات متكررة إلى مخدّم *Web* من أجل شحن جميع الملفات التي تكون برميح *applet*، إضافة إلى ذلك فإنّ أياً من هذه الملفات يكون بنمط غير مضغوط.

لذلك فعند تجميع الملفات التي تكون برميح *applet* في ملف وحيد بنمط *JAR*، ستحتاج إلى إجراء طلب مخدّم واحد فقط، كما أنّ عملية النقل ستكون أسرع بسبب الضغط. ويمكن ترميز كل مدخل في ملف *JAR* رقمياً مما يساعد في تحقيق أمن *security* هذا الملف بشكل ممتاز.

سلسلة العنصر *Object*

...serialization

وهي عبارة عن تقنية ممتازة تسمح لك بأخذ أي عنصر ينفذ الواجهة *Serializable*، وتقوم بتحويله إلى سلسلة بايتات يمكن إرجاعها فيما بعد إلى العنصر الأصلي. وتستخدم هذه التقنية عبر الشبكة بحيث تقوم وبشكل تلقائي بإجراء عملية التوافق بين مختلف أنظمة التشغيل. هذا يعني أنّ باستطاعتك إنشاء عنصر على حاسب *Windows* ثمّ سلسلته وإرساله عبر الشبكة إلى حاسب *Unix* حيث تتم إعادة بناءه

بشكل صحيح. لذلك لم يعد هنالك أي داع للقلق بشأن أشكال تمثيل المعطيات على الأجهزة المختلفة.

ولقد تمّ استخدام تقنية سلسلة العناصر *object serialization* لدعم تقنيتين رئيسيتين: الأولى تقنية *remote method invocation (RMI)* والتي تسمح للعناصر التي تعيش في حواسيب أخرى بالتصرّف وكأنها تعيش في حاسبك. أما الثانية فهي تقنية جافا بينز *Java Beans*، وعندما يتم استخدام البينز *beans* فإنّ معلومات الحالة له يتم توصيفها في وقت التصميم. ويجب أن يتم تخزين معلومات الحالة هذه واستردادها فيما بعد عند تشغيل البرنامج، حيث تقوم تقنية سلسلة العناصر بإنجاز هذه المهمة.

وتعتبر عملية سلسلة العناصر بسيطة، حيث تمّ تغيير العديد من صفوف المكتبة لتصبح مُسلسلة.

ومن أجل سلسلة عنصر، يجب عليك إنشاء عنصر من نمط *OutputStream* وتغليفه بعنصر *ObjectOutputStream*. عند هذه النقطة أنت بحاجة إلى استدعاء الطريقة *writeObject()* فقط، يتم بعدها سلسلة العنصر وإرساله إلى *OutputStream*.

الشيء المثير للانتباه في هذه العملية ليس حفظ صورة من عنصرك فقط، وإنما عملية تتبع جميع المؤشرات الموجودة في هذا العنصر وقيامك بحفظ هذه العناصر، وتتبع أيضاً جميع المؤشرات الموجودة في كل عنصر من هذه العناصر وهكذا...
يوضح المثال التالي كيفية استخدام هذه التقنية:

```
//: Worm.java
// Demonstrates object serialization in Java
1.1
import java.io.*;
class Data implements Serializable {
    private int i;
    Data(int x) { i = x; }
    public String toString() {
        return Integer.toString(i);
    }
}
```

سلسلة الرضا للمعلومات

```

}
public class Worm implements Serializable {
    // Generate a random int value:
    private static int r() {
        return (int)(Math.random() * 10);
    }
    private Data[] d = {
        new Data(r()), new Data(r()), new Data(r())
    };
    private Worm next;
    private char c;
    // Value of i == number of segments
    Worm(int i, char x) {
        System.out.println(" Worm constructor: " +
            i);
        c = x;
        if(--i > 0)
            next = new Worm(i, (char)(x + 1));
    }
    Worm() {
        System.out.println("Default constructor");
    }
    public String toString() {
        String s = ":" + c + "(";
        for(int i = 0; i < d.length; i++)
            s += d[i].toString();
        s += ")";
        if(next != null)
            s += next.toString();
        return s;
    }
    public static void main(String[] args) {
        Worm w = new Worm(6, 'a');
        System.out.println("w = " + w);
        try {
            ObjectOutputStream out =
                new ObjectOutputStream(
                    new FileOutputStream("worm.out"));

```

```

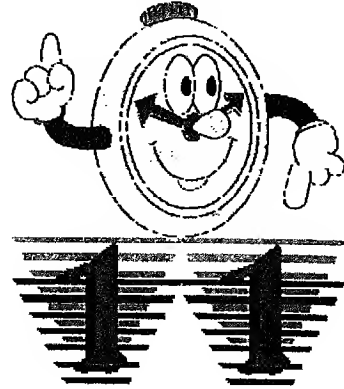
out.writeObject("Worm storage");
out.writeObject(w);
out.close(); // Also flushes output
ObjectInputStream in =
    new ObjectInputStream(
        new FileInputStream("worm.out"));
String s = (String)in.readObject();
Worm w2 = (Worm)in.readObject();
System.out.println(s + ", w2 = " + w2);
} catch(Exception e) {
    e.printStackTrace();
}
try {
    ByteArrayOutputStream bout =
        new ByteArrayOutputStream();
    ObjectOutputStream out =
        new ObjectOutputStream(bout);
    out.writeObject("Worm storage");
    out.writeObject(w);
    out.flush();
    ObjectInputStream in =
        new ObjectInputStream(
            new ByteArrayInputStream(
                bout.toByteArray()));
    String s = (String)in.readObject();
    Worm w3 = (Worm)in.readObject();
    System.out.println(s + ", w3 = " + w3);
} catch(Exception e) {
    e.printStackTrace();
}
}
} ///:~

```

في البرنامج السابق تمّ تحديد القيم الابتدائية لمصفوفة العناصر Data في الصف Worm بقيم عشوائية. كذلك تمت تسمية كل مقطع Worm بنمط Char بحيث يمكن توليده تلقائياً عند إجراء التوليد التلقائي لقائمة ارتباطات عناصر Worm. وعندما تقوم بإنشاء عنصر Worm، يتم إخبار الباني عن الزمن الذي تحتاج فيه إلى هذا العنصر. ولإنشاء المؤشر

التالي *Next* يتم استدعاء باني *Worm* بطول أقل بواحد وهكذا. أما مؤشر *Next* الأخير فيصبح *null* للدلالة على نهاية *Worm*.
 لاحظ بأن عملية سلسلة العناصر تتم ببساطة، فعندما يتم إنشاء العنصر *ObjectOutputStream* اعتماداً على أنماط دفع أخرى، تقوم الطريقة *writeObject()* بسلسلة العنصر.
 كما أن هناك كتلتين *try* متشابهتين: الكتلة الأولى تقوم بالكتابة والقراءة من ملف، أما الثانية فتقوم بالكتابة والقراءة من *ByteArray*.
 أما الخرج الناتج عن تنفيذ وحيد للبرنامج السابق فسيكون على الشكل:

```
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w2 =
:a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w3
:a(262):b(100):c(396):d(480):e(316):f(398)
```

عملية إنشاء واجهة مستخدم رسومية *GUI (Graphical User Interface)* تعتبر من المهام الأساسية التي تواجه المبرمجين. ولقد احتوى الإصدار *Java 1.0* على الأداة *AWT (Abstract Window Toolkit)* التي تساعد على بناء واجهات المستخدم الرسومية. لكنها كانت تحتوي على الكثير من نقاط الضعف من ناحية دعمها للخطوط أو تعاملها مع عناصر التحكم الأساسية.

سلسلة الرضا للمعلومات

ولقد تمّ حلّ الكثير من نقاط الضعف هذه في الإصدار *Java 1.1*، حيث أصبحت مكتبة *AWT* الجديدة غرضيّة التوجّه *Object-Oriented* بعد إضافة *Java Beans*.

أما في الإصدار *Java 1.2* فلقد تمّت إضافة الأداة *Swing* وإضافة مجموعة صفوف جديدة هي *JFC* (*Java Foundation Classes*) والتي سنتحدّث عنها في الفصل السابع عشر.

أما البرمجيات *Applets* فهي من أهم عناصر التصميم الأساسيّة في جافا، وهي عبارة عن برامج صغيرة تعمل داخل مستعرض الويب *Web Browser*.

البرميج الأساسي *Tha basic Applet* ...

يتمّ تجميع المكتبات في أغلب الأحيان وفقاً لعملها. وهناك نمط خاص من المكتبات اسمه نطاق التطبيق *application framework*، هدفها الأساسي مساعدتك في بناء التطبيقات، وذلك بتزويدك بمجموعة من الصفوف التي تحتوي على مجموعة من العناصر الأساسيّة التي تحتاجها لبناء التطبيقات.

ويتمّ بناء البرمجيات *Applets* باستخدام نطاق التطبيق *application framework*، حيث يتمّ التوريث من الصف *Applet*. وهناك العديد من الطرق الضروريّة لبناء البرمجيات الموضّحة في الجدول التالي:

الطريقة	عملها
<i>init()</i>	يتم استدعاؤها من أجل بدء عمل البرمج عند إنشائه.
<i>start()</i>	يتم استدعاؤها عند نقل البرمج إلى مشهد على مستعرض الويب، وذلك للسماح بإقلاع العمليات الأساسية للبرمج.
<i>paint()</i>	وهي جزء من الصف الأساسي <i>Component</i> . ويتم استدعاؤها كجزء من الطريقة <i>update()</i> لإنجاز بعض عمليات الرسم الخاصة على الكنافا <i>canvas</i> المتعلقة بالبرمج.
<i>stop()</i>	يتم استدعاؤها في كل مرة يتم فيها نقل البرمج خارج مشهد مستعرض الويب، وذلك للسماح للبرمج بإغلاق العمليات المكلفة. وتستدعى مباشرة قبل الطريقة <i>destroy()</i> .
<i>destroy()</i>	يتم استدعاؤها عند إلغاء تحميل البرمج من صفحة الويب، وذلك من أجل تحرير المصادر النهائية عند الانتهاء من استخدام البرمج.

يوضح المثال التالي كيفية إنشاء برمج بسيط:

```
//: Applet1.java
// Very simple applet
package c11;
import java.awt.*;
import java.applet.*;
public class Applet1 extends Applet {
    public void paint(Graphics g) {
        g.drawString("First applet", 10, 10);
    }
} ///:~
```

لاحظ بأن البرمج لا يحتاج إلى الطريقة الأساسية *main()*. ومن أجل تطبيق هذا البرنامج، عليك وضعه ضمن صفحة الويب ومن ثم مشاهدة الصفحة داخل المستعرض. أما من أجل وضع برمج داخل صفحة وب، فتحتاج إلى وضع علامات *tags* خاصة داخل

مصدر HTML الخاص بصفحة الويب، وذلك من أجل إخبار الصفحة عن طريقة تحميل

وتنفيذ البرمج، وهو ما يسمى بعلامة `applet`، وهي تشبه `Applet1` التالي:

```
<applet
code=Applet1
width=200
height=200>
</applet>
```

حيث يتم وضع اسم الملف `class` الذي يحتوي على البرمج كقيمة للوسيط `code`.

أما `width` و `height` فتحدد الحجم الابتدائي للبرمج.

أما البرنامج التالي فيوضح كيفية استخدام الطرق الأساسية الخاصة بالبرمجيات، والتي قمنا

بشرحها من قبل، حيث يقوم بإظهار عدد مرات استدعاء كل من هذه الطرق:

```
//: Applet3.java
// Shows init(), start() and stop() activities
import java.awt.*;
import java.applet.*;
public class Applet3 extends Applet {
    String s;
    int inits = 0;
    int starts = 0;
    int stops = 0;
    public void init() { inits++; }
    public void start() { starts++; }
    public void stop() { stops++; }
    public void paint(Graphics g) {
        s = "inits: " + inits +
            ", starts: " + starts +
            ", stops: " + stops;
        g.drawString(s, 10, 10);
    }
} ///:~
```

عندما نقوم باختبار هذا البرمج سنكتشف بأنك عندما تقوم بتصغير مستعرض الويب، أو

تغطيته بنافذة أخرى فلن تحتاج لاستدعاء الطريقتين `start()` و `stop()` ويتم

استدعاهما فقط عندما تنتقل إلى صفحة وب مختلفة ثم تعود بعد ذلك إلى الصفحة التي تحتوي على البرمج.

إنشاء زر...

تعتبر عملية إنشاء زر بسيطة جدا. فقط قم باستدعاء الباني `Button()` مع تحديد عنوان الزر. والزر `Button` عبارة عن مكون `component` يتم تشكيل النافذة الصغيرة الخاصة به تلقائيا. لذلك لن تكون بحاجة إلى رسم الزر بشكل صريح، بل يكفي وضعه فقط على النموذج وتركه يهتم برسم نفسه بشكل تلقائي.

يوضح المثال التالي كيفية رسم زر على البرمج:

```
//: Button1.java
// Putting buttons on an applet
import java.awt.*;
import java.applet.*;
public class Button1 extends Applet {
    Button
        b1 = new Button("Button 1"),
        b2 = new Button("Button 2");
    public void init() {
        add(b1);
        add(b2);
    }
} ///:~
```

كما تلاحظ هنا إنشاء زر `Button` لا يكفي، بل عليك استدعاء الطريقة `add()` الخاصة بالصف `Applet` أيضا لوضع الزر على نموذج البرمج.

التقاط حدث *Capturing an event* ...

عندما تقوم بترجمة وتنفيذ البرمج السابق، ستلاحظ بأنك لن تحصل على أي شيء عند النقر على الزر. لذلك عليك كتابة ترميزا ما لتحديد ما سيحدث عند النقر على الزر، وهذا من القواعد الأساسية الخاصة بالبرمجة المنقادة بالأحداث *Event-driven programming*.

والإصدار *Java1.0* لا يمتلك إلا عددا محددا من الأحداث التي يمكنك توليدها، بينما يستطيع الإصداران *Java1.1* و *Swing/JFC* توليد مجموعة كاملة من الأحداث. وعلى الرغم من أن نموذج الحدث الخاص بالإصدار *Java1.0* قد تم تغييره إلى حد كبير في الإصدار *Java1.1*، إلا أنه لا يزال يستخدم في البرمجيات البسيطة، وفي الأنظمة التي لم تدعم بعد الإصدار *Java1.1*.

ويمكن استخدام الطريقة *action()* من أجل تحديد مايتوجب عمله استجابة لحدث ما. في المثال التالي سنقوم بتعديل البرنامج السابق ليصبح بالإمكان التقاط النقرات على الزر ومعالجتها:

```
//: Button2.java
// Capturing button presses
import java.awt.*;
import java.applet.*;
public class Button2 extends Applet {
    Button
        b1 = new Button("Button 1"),
        b2 = new Button("Button 2");
    public void init() {
        add(b1);
        add(b2);
    }
    public boolean action(Event evt, Object arg)
    {
```

```

if(evt.target.equals(b1))
    getAppletContext().showStatus("Button 1");
else if(evt.target.equals(b2))
    getAppletContext().showStatus("Button 2");
// Let the base class handle it:
else
    return super.action(evt, arg);
return true; // We've handled it here
}
} ///:~

```

كي تستطيع رؤية الهدف *target*، اسأل العنصر *Event* عن العضو الهدف *target* المتعلق به، وبعد ذلك استخدم الطريقة *equals()* لمعرفة إن كان هذا العضو يساوي مؤشر العنصر الهدف الذي تبحث عنه.

وكما نرى في المثال السابق فإن الفعل *action* الناتج هو طباعة اسم الزر الذي تم ضغطه (من خلال الطريقة *ShowStatus()* الخاصة بالصف *Applet*، لذلك يمكن طلبها مباشرة دون استدعاء الطريقة *(getAppletContext())*).

الحقول النصية *Text Fields*...

الحقل النصي عبارة عن منطقة سطر واحد تسمح للمستخدم بإدخال نص وتحريره. والصف *TextField* مورث من الصف *TextComponent*، الذي يسمح لك باختيار نص، أو تحويل النص المختار إلى سلسلة محارف *String*، أو تحديد فيما إذا كان العنصر *TextField* قابلاً للتحرير.

والمثال البسيط التالي يوضح بعضاً من وظائف العنصر *TextField*:

```

//: TextField1.java
// Using the text field control
import java.awt.*;
import java.applet.*;
public class TextField1 extends Applet {
    Button
    b1 = new Button("Get Text"),

```

سلسلة الرضا للمعلومات

```

b2 = new Button("Set Text");
TextField
t = new TextField("Starting text", 30);
String s = new String();
public void init() {
    add(b1);
    add(b2);
    add(t);
}
public boolean action (Event evt, Object arg)
{
    if(evt.target.equals(b1)) {
        getAppletContext().showStatus(t.getText());
    };
    s = t.getSelectedText();
    if(s.length() == 0) s = t.getText();
    t.setEditable(true);
}
else if(evt.target.equals(b2)) {
    t.setText("Inserted by Button 2: " + s);
    t.setEditable(false);
}
// Let the base class handle it:
else
    return super.action(evt, arg);
return true; // We've handled it here
}
} ///:~

```

هناك عدة طرق لبناء عنصر *TextField*، إحدى هذه الطرق موضحة في المثال السابق، حيث يتم تحديد سلسلة محارف ابتدائية، وتحديد حجم الحقل بعدد المحارف. وعند النقر على الزر الأول سيتم جلب النص الذي تم تحديده بالفأرة، أو جلب كامل النص في الحقل، ووضع النتيجة في العنصر *String s*. كما يسمح بتحرير هذا الحقل أيضا.

أما عند النقر على الزر الثاني فسيتم وضع الرسالة المحددة بقيمة العنصر *s* في العنصر النصي، ومنع تحرير الحقل.

المناطق النصية Text Areas...

وتشبه كثيرا الصف *TextAreas* عدا أنها يمكن أن تحتوي على عدة أسطر، إضافة إلى العديد من الوظائف الخاصة بها. فيمكنك مثلا إلحاق نص أو إدراجه أو استبداله في موقع محدد. كما يمكنك إضافة شريط انزلاق أفقي أو عمودي إلى هذا العنصر. يوضح المثال التالي كيفية التعامل مع هذا النوع من العناصر:

```
//: TextAreal.java
// Using the text area control
import java.awt.*;
import java.applet.*;
public class TextAreal extends Applet {
    Button b1 = new Button("Text Area 1");
    Button b2 = new Button("Text Area 2");
    Button b3 = new Button("Replace Text");
    Button b4 = new Button("Insert Text");
    TextArea t1 = new TextArea("t1", 1, 30);
    TextArea t2 = new TextArea("t2", 4, 30);
    public void init() {
        add(b1);
        add(t1);
        add(b2);
        add(t2);
        add(b3);
        add(b4);
    }
    public boolean action (Event evt, Object arg)
    {
        if(evt.target.equals(b1))
            getAppletContext().showStatus(t1.getText());
        else if(evt.target.equals(b2)) {
            t2.setText("Inserted by Button 2");
            t2.appendText(": " + t1.getText());
            getAppletContext().showStatus(t2.getText());
        }
    }
}
```

```

    }
    else if(evt.target.equals(b3)) {
        String s = " Replacement ";
        t2.replaceText(s, 3, 3 + s.length());
    }
    else if(evt.target.equals(b4))
        t2.insertText(" Inserted ", 10);
    // Let the base class handle it:
    else
        return super.action(evt, arg);
    return true; // We've handled it here
}
} ///:~

```

هناك العديد من الطرق لبناء عناصر `TextArea`، إلا أن الطريقة الموضحة في المثال السابق تقوم على استخدام سلسلة محارف ابتدائية وتحديد عدد الأسطر وعدد الأعمدة. أما الأزرار المستخدمة فتقوم بعملية الحصول على نص، أو إلحاقه، أو استبداله، أو إدراج نصا جديدا.

...Labels المصاقات

يفيد هذا النوع من العناصر بوضع لصاقة (عنوان) على نموذج. وهي مفيدة بشكل خاص من أجل الحقول والمناطق النصية التي لا تمتلك لصاقات خاصة بها، كما أنها مفيدة عندما تحتاج لوضع معلومات نصية معينة على نموذجك كعنوان أو ماشابه ذلك. وباستطاعتك كما رأينا في المثال الأول من هذا الفصل استخدام الطريقة `drawString()` داخل `paint()` لوضع نص في موقع محدد. وعندما تقوم باستخدام عنصر `label`، يصبح بإمكانك ربط النص مع بعض المكونات الأخرى من خلال مدير المخطط `layout manager` (وهو ما سناقشه لاحقا في هذا الفصل).



ويمكنك إنشاء لصاقة فارغة، أو لصاقة مع نص مبدئي، أو لصاقة مع محاذاة معينة (CENTER, LEFT, Right). وتستطيع تغيير اللصاقة وتغيير المحاذاة فيها باستخدام `setText()` و `setAlignment()`.
والمثال التالي يوضح ما تستطيع عمله مع اللصاقات:

```
//: Label1.java
// Using labels
import java.awt.*;
import java.applet.*;
public class Label1 extends Applet {
    TextField t1 = new TextField("t1", 10);
    Label labl1 = new Label("TextField t1");
    Label labl2 = new Label(" ");
    Label labl3 = new Label(" ",
        Label.RIGHT);
    Button b1 = new Button("Test 1");
    Button b2 = new Button("Test 2");
    public void init() {
        add(labl1); add(t1);
        add(b1); add(labl2);
        add(b2); add(labl3);
    }
    public boolean action (Event evt, Object arg)
    {
        if(evt.target.equals(b1))
            labl2.setText("Text set into Label");
        else if(evt.target.equals(b2)) {
            if(labl3.getText().trim().length() == 0)
                labl3.setText("labl3");
            if(labl3.getAlignment() == Label.LEFT)
                labl3.setAlignment(Label.CENTER);
            else if(labl3.getAlignment() == Label.CENTER)
                labl3.setAlignment(Label.RIGHT);
            else if(labl3.getAlignment() ==
                Label.RIGHT)
                labl3.setAlignment(Label.LEFT);
        }
    }
}
```

```

else
    return super.action(evt, arg);
return true;
}
} ///:~

```

الاستخدام الأول للصاق هو الاستخدام التقليدي وذلك مع عناصر `TextField` أو `TextArea`. أما في الجزء الثاني من هذا المثال فيتم إدراج نص في الصاق `lab12` عند النقر على الزر "Test 1". أما عند النقر على "Test 2" فيتم اختبار وجود محارف في الصاق `lab13` وإدراج النص "lab13". وفي نهاية هذا الجزء يتم اختبار محاذاة النص ضمن الصاق `lab13` مع تغييرها وذلك كمثال على استخدام الطريقة `setAlignment()`.

ويجب الانتباه إلى عدم إمكانية إنشاء لصاق فارغة ووضع النص بعد ذلك لأنه لا يمكن وضع نص في لصاق عديمة العرض. لذلك وفي هذا المثال، ومن أجل إنشاء لصاق فارغة، قمنا بتعبئتها بفراغات كيلا تصبح عديمة الطول.

صناديق التحقق Check Boxes...

يمكنك بسهولة إنشاء عنصر من نمط `CkeckBox` باستخدام بـان `constructor` يأخذ عنوان الصندوق كوسيط. وبإمكانك الحصول على حالة صندوق التحقق وتغييرها، وبإمكانك أيضا الحصول على عنوان الصندوق وتغييره.

وبعد أن يتم إنشاء وتحديد صندوق تحقق، يتم تحديد الحدث الموافق تماما كما في الأزرار. سنقوم في المثال التالي باستخدام عنصر `TextArea` لتعداد جميع صناديق التحقق التي تم تفعيلها:

```

//: CheckBox1.java
// Using check boxes
import java.awt.*;
import java.applet.*;
public class CheckBox1 extends Applet {
    TextArea t = new TextArea(6, 20);

```



```

Checkbox cb1 = new Checkbox("Check Box 1");
Checkbox cb2 = new Checkbox("Check Box 2");
Checkbox cb3 = new Checkbox("Check Box 3");
public void init() {
    add(t); add(cb1); add(cb2); add(cb3);
}
public boolean action (Event evt, Object arg)
{
    if(evt.target.equals(cb1))
        trace("1", cb1.getState());
    else if(evt.target.equals(cb2))
        trace("2", cb2.getState());
    else if(evt.target.equals(cb3))
        trace("3", cb3.getState());
    else
        return super.action(evt, arg);
    return true;
}
void trace(String b, boolean state) {
    if(state)
        t.appendText("Box " + b + " Set\n");
    else
        t.appendText("Box " + b + " Cleared\n");
}
} ///:~

```

تقوم الطريقة `trace()` بإرسال اسم صندوق التحقق `CheckBox` الذي تم اختياره، وحالة هذا الصندوق إلى عنصر `TextArea` ضمن الطريقة `appendText()` وسترى قائمة بصناديق التحقق التي تم اختيارها وحالة كل منها.

أزرار الراديو **Radio Buttons**...

ليس هناك صف خاص لتمثيل أزرار الراديو ضمن AWT، بل يتم استخدام الصف *CheckBox*. ومن أجل وضع عنصر *CheckBox* ضمن مجموعة أزرار راديو، عليك استخدام بان خاص يأخذ عنصرا من نمط *CheckboxGroup* كوسيط. ولا يحتوي باني الصف *CheckboxGroup* على أي وسيط، لأن السبب الأساسي لوجوده هو تجميع بعض أزرار التحقق *Checkbox* على شكل مجموعة أزرار راديو. ويجب أن تكون أحد أزرار التحقق في المجموعة في حالة *true* قبل محاولة إظهار هذه المجموعة، وإلا فستحصل على استثناء في وقت التنفيذ. وعندما تحاول تحديد أكثر من زر على القيمة *true* فسيتم تحديد هذه القيمة على الزر الأخير فقط. سنقوم في المثال التالي بتوضيح كيفية استخدام أزرار الراديو، ولاحظ أنه يتم النقاط أحداث زر الراديو بشكل مشابه تماما لما رأيناه سابقا:

```
//: RadioButton1.java
// Using radio buttons
import java.awt.*;
import java.applet.*;
public class RadioButton1 extends Applet {
    TextField t =
        new TextField("Radio button 2", 30);
    CheckboxGroup g = new CheckboxGroup();
    Checkbox
        cb1 = new Checkbox("one", g, false),
        cb2 = new Checkbox("two", g, true),
        cb3 = new Checkbox("three", g, false);
    public void init() {
        t.setEditable(false);
        add(t);
        add(cb1); add(cb2); add(cb3);
    }
    public boolean action (Event evt, Object arg)
    {
        if (evt.target.equals(cb1))
```



```

        t.setText("Radio button 1");
    else if(evt.target.equals(cb2))
        t.setText("Radio button 2");
    else if(evt.target.equals(cb3))
        t.setText("Radio button 3");
    else
        return super.action(evt, arg);
    return true;
}
} ///:~

```

من أجل إظهار الحالة قمنا باستخدام حقلا نصيا، حيث تم تحديد هذا الحقل على أنه غير قابل للتعديل لأنه سيستخدم لإظهار المعطيات فقط. لاحظ بدء الحقل بالقيمة "Radio button 2" لأنها القيمة الابتدائية لمجموعة أزرار الراديو.

الوائم المتبدلية Drop-Down Lists...

وهي عبارة عن طريقة لإجبار المستخدم على اختيار قيمة وحيدة من مجموعة خيارات معطاة. وتحتوي لغة جافا على الصندوق *Choice*، وهو لا يشبه صندوق السرد والتحرير *combo box* في *Windows*، لأنه لا يمكنك إلا اختيار قيمة وحيدة فقط من اللائحة، كما أنه لا يمكنك من كتابة قيمة إضافية. في المثال التالي سنستخدم الصندوق *Choice* مع مجموعة قيم محددة، وستتم إضافة قيم أخرى إليه عند النقر على زر:

```

//: Choicer1.java
// Using drop-down lists
import java.awt.*;
import java.applet.*;
public class Choicer1 extends Applet {
    String[] description = { "Ebullient",
        "Obtuse",
        "Recalcitrant", "Brilliant", "Somnescent",
        "Timorous", "Florid", "Putrescent" };
}

```

```

TextField t = new TextField(30);
Choice c = new Choice();
Button b = new Button("Add items");
int count = 0;
public void init() {
    t.setEditable(false);
    for(int i = 0; i < 4; i++)
        c.addItem(description[count++]);
    add(t);
    add(c);
    add(b);
}
public boolean action (Event evt, Object arg)
{
    if(evt.target.equals(c))
        t.setText("index: " +
            c.getSelectedIndex()
            + " " + (String)arg);
    else if(evt.target.equals(b)) {
        if(count < description.length)
            c.addItem(description[count++]);
    }
    else
        return super.action(evt, arg);
    return true;
}
} ///:~

```

يقوم العنصر `TextField` بإظهار الرقم التسلسلي للعنصر الذي تم اختياره، إضافة إلى التمثيل بنمط `String` للوسيط الثاني في الطريقة `action()`، وهو في هذا المثال السلسلة التي تم اختيارها.



صناديق اللائحة *List Boxes*...

تختلف صناديق اللائحة كليا عن صناديق الاختيار *Choice*. فبينما تتدلى لائحة عند تفعيل صندوق الاختيار، يحتل صندوق اللائحة *List Box* عدد محدد من الأسطر على النافذة. إضافة إلى ذلك تسمح لك اللوائح باختيار عدة اختيارات، فإذا قمت بالنقر على أكثر من عنصر في القائمة، يبقى العنصر الأصلي غامقا وبإمكانك اختيار أي عدد من العناصر. إذا أردت إظهار عناصر اللائحة، قم فقط باستدعاء الطريقة *getSelectedItems()* التي تعطيك مصفوفة سلاسل حرفية *String* تم اختيارها. يمكنك إلغاء عنصر من المجموعة التي تم اختيارها بإعادة النقر عليه. المشكلة الأساسية في اللوائح هي أنك تحتاج للنقر المزدوج بالفأرة لاختيار عنصر من اللائحة، لأن النقر المفرد يساعدك على إضافة أو حذف عناصر من مجموعة محددة، أما النقر المزدوج فيؤدي إلى استدعاء الطريقة *action()*. يوضح المثال التالي كيفية التعامل مع صناديق اللائحة:

```
//: List1.java
// Using lists with action()
import java.awt.*;
import java.applet.*;
public class List1 extends Applet {
    String[] flavors = { "Chocolate",
        "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie" };
    // Show 6 items, allow multiple selection:
    List lst = new List(6, true);
    TextArea t = new TextArea(flavors.length,
        30);
    Button b = new Button("test");
    int count = 0;
    public void init() {
        t.setEditable(false);
        for(int i = 0; i < 4; i++)
```

```

        lst.addItem(flavors[count++]);
        add(t);
        add(lst);
        add(b);
    }
    public boolean action (Event evt, Object arg)
    {
        if(evt.target.equals(lst)) {
            t.setText("");
            String[] items = lst.getSelectedItems();
            for(int i = 0; i < items.length; i++)
                t.appendText(items[i] + "\n");
        }
        else if(evt.target.equals(b)) {
            if(count < flavors.length)
                lst.addItem(flavors[count++], 0);
        }
        else
            return super.action(evt, arg);
        return true;
    }
} ///:~

```

كما تلاحظ في البرنامج السابق، فعندما تقوم بالنقر على الزر *b*، تتم إضافة عناصر إلى رأس القائمة.

التحكم بالتخطيط Controlling layout...layout

تختلف الطريقة التي تقوم فيها بوضع العناصر على نموذج في جافا عن أي نظام GUI آخر قمت بالتعامل معه: فهي أولا مرمزة بشكل كامل ولا توجد مصادر تتحكم بتوضع المكونات. وثانيا يتم التحكم بطريقة وضع المكونات على نموذج من خلال مدير التخطيط *layout manager*. ويختلف حجم وشكل وتوضع المكونات بشكل ملحوظ من تخطيط لآخر. إضافة إلى ذلك يقوم مدير التخطيط *layout manager* بإجراء عملية لملائمة أبعاد البرمج أو نافذة التطبيق، بحيث إذا تم تغيير أبعاد النافذة بالتالي سيتغير حجم وشكل وتوضع المكونات.

وكلا الصفان *Applet* و *Frame* مشتقان من الصف *Container*. وعمل الصف *Container* هو احتواء وإظهار عناصر *Components*، وهو يحتوي على الطريقة *setLayout()* التي تسمح لك باختيار تخطيطات مختلفة. سنقوم فيما يلي باستكشاف مديري التخطيط العديدين الذين يمكن التعامل معهم:

...FlowLayout

وهو التخطيط الافتراضي، ويتم فيه توضع المكونات على النموذج من اليسار إلى اليمين حتى يصبح الجزء العلوي ممتلئا، عندها ينتقل سطر نحو الأسفل ويتابع بنفس الطريقة. سنقوم في المثال التالي بتحديد مدير التخطيط ليأخذ القيمة *FlowLayout* ووضع الأزرار على النموذج:

```
//: FlowLayout1.java
// Demonstrating the FlowLayout
import java.awt.*;
import java.applet.*;
```

سلسلة الرضا للمعلومات

```
public class FlowLayout1 extends Applet {
    public void init() {
        setLayout(new FlowLayout());
        for(int i = 0; i < 20; i++)
            add(new Button("Button " + i));
    }
} ///:~
```

ستلاحظ هنا بأنه سيتم إعادة جميع المكونات إلى حجمها الأصغر.

...BorderLayout

يمتلك مدير التخطيط أربع مناطق حدودية ومنطقة مركزية. وعندما نقوم بإضافة أي شيء إلى لوحة ما باستخدام *BorderLayout*، يتوجب عليك استخدام الطريقة *add()* التي تأخذ عنصر *String* كوسيط أول يأخذ إحدى القيم التالية: "North"، "South"، "East"، "West"، "Center".
لنأخذ المثال التالي على هذا النمط من التخطيط:

```
//: BorderLayout1.java
// Demonstrating the BorderLayout
import java.awt.*;
import java.applet.*;
public class BorderLayout1 extends Applet {
    public void init() {
        int i = 0;
        setLayout(new BorderLayout());
        add("North", new Button("Button " + i++));
        add("South", new Button("Button " + i++));
        add("East", new Button("Button " + i++));
        add("West", new Button("Button " + i++));
        add("Center", new Button("Button " + i++));
    }
} ///:~
```



...GridLayout

يسمح لك هذا التخطيط ببناء شبكة مكونات، وعندما تقوم بإضافة هذه المكونات إلى الشبكة، يتم وضعها من اليسار إلى اليمين ومن الأعلى إلى الأسفل ضمن الشبكة. ويجب أن تحدد، ضمن الباني *constructor*، عدد الأسطر والأعمدة التي تحتاجها في الشبكة. انظر البرنامج التالي:

```
//: GridLayout1.java
// Demonstrating the FlowLayout
import java.awt.*;
import java.applet.*;
public class GridLayout1 extends Applet {
    public void init() {
        setLayout(new GridLayout(7,3));
        for(int i = 0; i < 20; i++)
            add(new Button("Button " + i));
    }
} ///:~
```

في هذه الحالة يوجد ٢١ شقا *slots* و ٢٠ زرا فقط. لذلك ترك الشق الأخير فارغا.

...CardLayout

يحتوي هذا التخطيط على مجموعة من بطاقات الحوار، وعند النقر على زر *tab* يتم الانتقال إلى صندوق حوار مختلف.

هناك الكثير من الصعوبات التي تواجهها عند استخدامك هذا التخطيط، لذلك سنقوم في الفصل الأخير بإعطائك طريقة أفضل لإنشاء مثل هذا النمط من التخطيط.

بدائل الطريقة (*action*)...

عند حصول حدث ما على عنصر، يتم استدعاء الطريقة (*handleEvent*) تلقائياً حيث يتم إنشاء عنصر *Event* وتمثيله إلى هذه الطريقة. وتوجد أيضاً (بالإضافة إلى الطريقة (*action*)) ثلاث مجموعات أخرى من الطرق التي يتم استدعاؤها عند حدوث حدث ما. وإذا أردت النقاط أنماط معينة من الأحداث (لوحة المفاتيح، الفأرة وغيرها) فما عليك إلا إبطال *override* الطريقة الموافقة. جميع هذه الطرق معرفة ضمن الصف الأساسي *Component*، لذلك فهي متاحة لجميع عناصر التحكم التي قد تضعها ضمن نموذجك.

يوضح الجدول التالي جميع أنماط الطرق التي يمكنك استخدامها كبداية للطريقة

action():

الطريقة	متى يتم استدعاؤها
<i>action(Event evt, Object what)</i>	عند حصول الحدث التقليدي على مكون ما.
<i>KeyDown(Event evt, int key)</i>	عند ضغط مفتاح، وذلك عندما يتم التركيز على مكون.
<i>KeyUp(Event evt, int key)</i>	عند إفلات مفتاح، وذلك عندما يتم التركيز على مكون.
<i>lostFocus(Event evt, Object what)</i>	عند انتقال التركيز خارج الهدف.
<i>gotFocus(Event evt, Object what)</i>	عند انتقال التركيز إلى الهدف.
<i>mouseDown(Event evt, int x, int y)</i>	عند ضغط الفأرة على المكون المحدد بالإحداثيات <i>x</i> و <i>y</i> .
<i>mouseUp(Event evt, int x, int y)</i>	عند إفلات الفأرة على المكون المحدد بالإحداثيات <i>x</i> و <i>y</i> .



عند تحريك الفأرة فوق العنصر.	<code>mouseMove (Event evt, int x, int y)</code>
عند سحب الفأرة بعد حدوث <code>mouseDown</code> على المكون.	<code>mouseDrag (Event evt, int x, int y)</code>
لم تكن الفأرة فوق المكون من قبل، لكنها الآن كذلك.	<code>mouseEnter (Event evt, int x, int y)</code>
كانت الفأرة المستخدمة فوق المكون، وخرجت الآن منه.	<code>mouseExit (Event evt, int x, int y)</code>

يمكنك ملاحظة أن كل طريقة تتلقى حدث `Event` مع بعض المعلومات التي ستحتاجها عند معالجتك وضعها خاصا.

يبين المثال التالي كيفية استخدام الطرق السابقة بشكل بسيط وواضح:

```

//: AutoEvent.java
// Alternatives to action()
import java.awt.*;
import java.applet.*;
import java.util.*;
class MyButton extends Canvas {
    AutoEvent parent;
    Color color;
    String label;
    MyButton(AutoEvent parent,
              Color color, String label) {
        this.label = label;
        this.parent = parent;
        this.color = color;
    }
    public void paint(Graphics g) {
        g.setColor(color);
        int rnd = 30;
        g.fillRoundRect(0, 0, size().width,
                        size().height, rnd, rnd);
        g.setColor(Color.black);
        g.drawRoundRect(0, 0, size().width,

```

```

size().height, rnd, rnd);
FontMetrics fm = g.getFontMetrics();
int width = fm.stringWidth(label);
int height = fm.getHeight();
int ascent = fm.getAscent();
int leading = fm.getLeading();
int horizMargin = (size().width - width)/2;
int verMargin = (size().height - height)/2;
g.setColor(Color.white);
g.drawString(label, horizMargin,
               verMargin + ascent + leading);
}
public boolean keyDown(Event evt, int key) {
    TextField t =
        (TextField)parent.h.get("keyDown");
    t.setText(evt.toString());
    return true;
}
public boolean keyUp(Event evt, int key) {
    TextField t =
        (TextField)parent.h.get("keyUp");
    t.setText(evt.toString());
    return true;
}
public boolean lostFocus(Event evt, Object w) {
    TextField t =
        (TextField)parent.h.get("lostFocus");
    t.setText(evt.toString());
    return true;
}
public boolean gotFocus(Event evt, Object w) {
    TextField t =
        (TextField)parent.h.get("gotFocus");
    t.setText(evt.toString());
    return true;
}
public boolean
mouseDown(Event evt,int x,int y) {

```



```

TextField t =
    (TextField)parent.h.get("mouseDown");
t.setText(evt.toString());
return true;
}
public boolean
mouseDrag(Event evt,int x,int y) {
    TextField t =
        (TextField)parent.h.get("mouseDrag");
    t.setText(evt.toString());
    return true;
}
public boolean
mouseenter(Event evt,int x,int y) {
    TextField t =
        (TextField)parent.h.get("mouseenter");
    t.setText(evt.toString());
    return true;
}
public boolean
mouseExit(Event evt,int x,int y) {
    TextField t =
        (TextField)parent.h.get("mouseExit");
    t.setText(evt.toString());
    return true;
}
public boolean
mousemove(Event evt,int x,int y) {
    TextField t =
        (TextField)parent.h.get("mousemove");
    t.setText(evt.toString());
    return true;
}
public boolean mouseUp(Event evt,int x,int y) {
    TextField t =
        (TextField)parent.h.get("mouseUp");
    t.setText(evt.toString());
    return true;
}

```

```

}
}
public class AutoEvent extends Applet {
    Hashtable h = new Hashtable();
    String[] event = {
        "keyDown", "keyUp", "lostFocus",
        "gotFocus", "mouseDown", "mouseUp",
        "mouseMove", "mouseDrag", "mouseenter",
        "mouseExit"
    };
    MyButton
        b1 = new MyButton(this, Color.blue, "test1"),
        b2 = new MyButton(this, Color.red, "test2");
    public void init() {
        setLayout(new GridLayout(event.length+1,2));
        for(int i = 0; i < event.length; i++) {
            TextField t = new TextField();
            t.setEditable(false);
            add(new Label(event[i], Label.CENTER));
            add(t);
            h.put(event[i], t);
        }
        add(b1);
        add(b2);
    }
} ///:~

```

كما تلاحظ تبدأ الطريقة `paint()` بتعبئة مستطيل دائري `round rectangle` مع تعبئة ألوان الأزوار، ترسم حولها بعد ذلك خطاً أسود. لاحظ استخدام الطريقة `size()` لتحديد طول وعرض المكون. تقوم بعد ذلك الطريقة `paint()` بإجراء الكثير من الحسابات من أجل وضع عنوان الزر في مركزه. بالطبع لن تتمكن من فهم كيفية عمل الطرق `keyDown()` و `keyUp()` ... حتى تقوم بإلقاء نظرة مطولة على الصف `AutoEvent`. يحتوي هذا الصف على الصف `Hashtable` من أجل احتواء سلاسل المحارف التي تمثل نمط الحدث، وعلى الصف `TextField` الذي سيحتوي على معلومات عن هذا الحدث.



بالطبع، باستطاعتك إنشاء هذه العناصر بشكل ساكن *statically* بدلا من وضعها في *Hashtable*، لكن أظن بأن هذه الطريقة أسهل وأفضل، خاصة عندما تحتاج لإضافة أو حذف نمط حدث من *AutoEvent*، لأنك ستحتاج ببساطة إلى إضافة أو حذف سلسلة محارف من المصفوفة *event* وسيتم تغيير كل شيء بعد ذلك بشكل تلقائي. أما المكان الذي سيتم البحث فيه عن السلاسل فسيكون ضمن الطرق *keyDown()* و *keyUp() ...* في الصف *MyButton*. وتستخدم كل من هذه الطرق مؤشرا إلى العنصر *parent* للانتقال إلى النافذة الأم. وبما أن العنصر الأب موجود في *AutoEvent*، فسيحتوي على العنصر *h Hashtable* وعلى الطريقة *get()* التي سترجع مؤشرا إلى *Object* عند تزويدها بسلسلة المحارف *String* المناسبة. ويتم بعد ذلك تحويل العنصر *Event* إلى التمثيل الموافق له على شكل سلسلة محارف *String*، حيث سيتم إظهاره ضمن *TextField*.

إنشاء نوافذ التطبيقات *Windowed Applications*

في كثير من الأحيان نحتاج إلى إنشاء برنامج نافذة *windowed program* يقوم بعمل شيء مختلف عند إنشاء موقع على صفحة وب. ويمكن أن تمتلك نافذة تطبيق على قوائم *menus* وصناديق حوار *dialog boxes*، وهو ما لا نستطيع عمله مع البرمجيات.

القوائم *Menus*

ليس بإمكانك وضع قائمة مباشرة على بريمج *applet*، بينما يمكنك القيام بذلك ضمن التطبيقات *applications*.

وتوجد أربعة أنماط مشتقة من الصف المجرد *MenuComponents*، وهي:

١. *MenuBar*: شريط قائمة على عنصر إطار *Frame* خاص.
٢. *Menu*: من أجل القوائم المتدلية *drop-down menus* والقوائم الفرعية *submenus*.
٣. *MenuItem*: لتمثيل عنصر وحيد على قائمة.
٤. *CheckboxMenuItem*: وهو مشتق من *MenuItem* وتعطي علامة تحقق لتحديد فيما إذا تم اختيار عنصر قائمة أم لا. كمثال على إنشاء القوائم سنقوم بإنشاء البرنامج التالي:

```
//: Menu1.java
// Menus work only with Frames.
// Shows submenus, checkbox menu items
// and swapping menus.
import java.awt.*;
public class Menu1 extends Frame {
```

```

String[] flavors = { "Chocolate",
    "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
TextField t = new TextField("No flavor", 30);
MenuBar mbl = new MenuBar();
Menu f = new Menu("File");
Menu m = new Menu("Flavors");
Menu s = new Menu("Safety");
// Alternative approach:
CheckboxMenuItem[] safety = {
    new CheckboxMenuItem("Guard"),
    new CheckboxMenuItem("Hide")
};
MenuItem[] file = {
    new MenuItem("Open"),
    new MenuItem("Exit")
};
// A second menu bar to swap to:
MenuBar mb2 = new MenuBar();
Menu fooBar = new Menu("fooBar");
MenuItem[] other = {
    new MenuItem("Foo"),
    new MenuItem("Bar"),
    new MenuItem("Baz"),
};
Button b = new Button("Swap Menus");
public Menu1() {
    for(int i = 0; i < flavors.length; i++) {
        m.add(new MenuItem(flavors[i]));
        // Add separators at intervals:
        if((i+1) % 3 == 0)
            m.addSeparator();
    }
    for(int i = 0; i < safety.length; i++)
        s.add(safety[i]);
    f.add(s);
}

```

```

for(int i = 0; i < file.length; i++)
    f.add(file[i]);
mbl.add(f);
mbl.add(m);
setMenuBar(mbl);
t.setEditable(false);
add("Center", t);
// Set up the system for swapping menus:
add("North", b);
for(int i = 0; i < other.length; i++)
    fooBar.add(other[i]);
mb2.add(fooBar);
}
public boolean handleEvent(Event evt) {
    if(evt.id == Event.WINDOW_DESTROY)
        System.exit(0);
    else
        return super.handleEvent(evt);
    return true;
}
public boolean action(Event evt, Object arg)
{
    if(evt.target.equals(b)) {
        MenuBar m = getMenuBar();
        if(m == mbl) setMenuBar(mb2);
        else if (m == mb2) setMenuBar(mbl);
    }
    else if(evt.target instanceof MenuItem) {
        if(arg.equals("Open")) {
            String s = t.getText();
            boolean chosen = false;
            for(int i = 0; i < flavors.length; i++)
                if(s.equals(flavors[i]), chosen = true;
            if(!chosen)
                t.setText("Choose a flavor first!");
            else
                t.setText("Opening "+ s +". Mmm, mm!");
        }
    }
}

```

```

else if(evt.target.equals(file[1]))
    System.exit(0);
// CheckboxMenuItems cannot use String
// matching; you must match the target:
else if(evt.target.equals(safety[0]))
    t.setText("Guard the Ice Cream! " +
    "Guarding is " + safety[0].getState());
else if(evt.target.equals(safety[1]))
    t.setText("Hide the Ice Cream! " +
    "Is it cold? " + safety[1].getState());
else
    t.setText(arg.toString());
}
else
    return super.action(evt, arg);
return true;
}
public static void main(String[] args) {
    Menu f = new Menu();
    f.resize(300,200);
    f.show();
}
} ///:~

```

كما تلاحظ في هذا البرنامج، فقد قمنا بوضع عناصر القائمة في مصفوفات، وتقلنا ضمن كل مصفوفة مع استدعاء الطريقة `add()` ضمن حلقة `for`. قمنا أيضا بإنشاء عناصر `CheckboxMenuItems` في مصفوفة مؤشرات أسمينها `safety`، وأجرينا نفس الشيء بالنسبة للمصفوفات `file` و `other`. وفي هذا البرنامج تم إنشاء عنصرين من نمط `MenuBar` للبرهان على أنه يمكن التنقل بين أشرطة القوائم عندما يكون البرنامج في حالة تنفيذ. يمكنك ملاحظة كيف أن كل عنصر `MenuBar` مؤلف من عدة عناصر `Menus`، وكل عنصر `Menu` مؤلف من عدة عناصر `MenuItem` و `CheckboxMenuItem` وحتى عناصر `Menus` أخرى. وعندما يتم تجميع عناصر `MenuBar` يكون من الممكن تثبيتها في البرنامج الحالي باستخدام الطريقة `setMenuBar()`.

لاحظ أيضا بأنه عند ضغط الزر، يقوم بالتحقق لمعرفة أي قائمة مثبتة حاليا باستخدام `getMenuBar()`، ويقوم بعدها بوضع بقية أشرطة القوائم في مكانها. وقد يبدو لك للوهلة الأولى بأنه من المنطقي وجود قائمة ما في أكثر من شريط قوائم. لكن مع الأسف وعند محاولتك القيام بذلك سيكون سلوك البرنامج غريبا بشكل غير متوقع. يوضح هذا المثال أيضا كل ما تحتاجه لإنشاء تطبيق بدلا من بريمج. فبدلا من التوريث من الصف `Applet`، سيتم التوريث من الصف `Frame`. وبدلا من استخدام الطريقة `init()` لبدء العناصر، ستقوم بإنشاء بان خاص لصفك. ستقوم أخيرا بإنشاء الطريقة `main()`، وستقوم ضمنها ببناء عنصر من النمط الجديد وتغيير حجمه، ثم استدعاء `show()`. يختلف الأمر هنا عما رأيناه في البرمجيات في أمور بسيطة، لكنك ستحصل على تطبيق بنافذة خاصة وستحصل فيها على قوائم أيضا.

صناديق الحوار Dialog Boxes...

كما تعرف فإن صندوق الحوار عبارة عن نافذة تظهر خارج نافذة أخرى. والهدف منها هو معالجة بعض الأمور الخاصة كي لا تتراكم هذه التفاصيل على النافذة الأصلية. وكما تعلم فإن صناديق الحوار تستخدم بكثرة في تطبيقات النوافذ، لكنها نادرة الاستخدام في البرمجيات.

لإنشاء صندوق حوار يجب التوريث من الصف `Dialog`، وهو نمط آخر من الصف `Window` أو الصف `Frame`.

وبعكس الصف `Frame`، لا يمكن للصف `Dialog` أن يمتلك شريط قوائم، أو يقوم بتغيير المؤشرة `cursor`، لكن ماعدا ذلك فهما متشابهان.

ولكل صندوق حوار مدير تخطيط `layout manager` (ويأخذ `BorderLayout` كتخطيط افتراضي) حيث يمكن استخدام `action()` أو `handleEvent()` لمعالجة الأحداث. الشيء الوحيد الذي ستجده مختلفا في الطريقة `handleEvent()` هو أنه عند حصول الحدث `WINDOW_DESTROY` فلن تحتاج

لإغلاق تطبيقك، وإنما ستحتاج إلى تحرير المصادر التي استخدمتها نافذة صندوق الحوار فقط وذلك باستدعاء الطريقة `dispose()`.

سنقوم في المثال التالي بإنشاء صندوق حوار مؤلف من شبكة (باستخدام التخطيط `GridLayout`) من أنواع الأزرار الخاصة المعرفة كصف `ToggleButton`. وكل زر يقوم برسم إطار حول نفسه، وفقا للحالة التي يعيش فيها!! فهو يبدأ بفراغات `blank`، أما بعدها، واعتمادا على من يقوم بنقره، سيتغير إلى "x" أو "o" عن طريق التبديل بينهما عند كل نقرة على الزر:

```
//: ToeTest.java
// Demonstration of dialog boxes
// and creating your own components
import java.awt.*;
class ToeButton extends Canvas {
    int state = ToeDialog.BLANK;
    ToeDialog parent;
    ToeButton(ToeDialog parent) {
        this.parent = parent;
    }
    public void paint(Graphics g) {
        int x1 = 0;
        int y1 = 0;
        int x2 = size().width - 1;
        int y2 = size().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int wide = x2/2;
        int high = y2/2;
        if(state == ToeDialog.XX) {
            g.drawLine(x1, y1, x1 + wide, y1 + high);
            g.drawLine(x1, y1 + high, x1 + wide, y1);
        }
        if(state == ToeDialog.OO) {
            g.drawOval(x1, y1, x1+wide/2, y1+high/2);
        }
    }
}
```

```

public boolean
mouseDown(Event evt, int x, int y) {
    if(state == ToeDialog.BLANK) {
        state = parent.turn;
        parent.turn= (parent.turn == ToeDialog.XX ?
            ToeDialog.OO : ToeDialog.XX);
    }
    else
        state = (state == ToeDialog.XX ?
            ToeDialog.OO : ToeDialog.XX);
    repaint();
    return true;
}
}
class ToeDialog extends Dialog {
    // w = number of cells wide
    // h = number of cells high
    static final int BLANK = 0;
    static final int XX = 1;
    static final int OO = 2;
    int turn = XX; // Start with x's turn
    public ToeDialog(Frame parent, int w, int h)
    {
        super(parent, "The game itself", false);
        setLayout(new GridLayout(w, h));
        for(int i = 0; i < w * h; i++)
            add(new ToeButton(this));
        resize(w * 50, h * 50);
    }
    public boolean handleEvent(Event evt) {
        if(evt.id == Event.WINDOW_DESTROY)
            dispose();
        else
            return super.handleEvent(evt);
        return true;
    }
}
public class ToeTest extends Frame {

```



```

TextField rows = new TextField("3");
TextField cols = new TextField("3");
public ToeTest() {
    setTitle("Toe Test");
    Panel p = new Panel();
    p.setLayout(new GridLayout(2,2));
    p.add(new Label("Rows", Label.CENTER));
    p.add(rows);
    p.add(new Label("Columns", Label.CENTER));
    p.add(cols);
    add("North", p);
    add("South", new Button("go"));
}
public boolean handleEvent(Event evt) {
    if(evt.id == Event.WINDOW_DESTROY)
        System.exit(0);
    else
        return super.handleEvent(evt);
    return true;
}
public boolean action(Event evt, Object arg)
{
    if(arg.equals("go")) {
        Dialog d = new ToeDialog(
            this,
            Integer.parseInt(rows.getText()),
            Integer.parseInt(cols.getText()));
        d.show();
    }
    else
        return super.action(evt, arg);
    return true;
}
public static void main(String[] args) {
    Frame f = new ToeTest();
    f.resize(200,100);
    f.show();
}

```

} ///:~

كما تلاحظ في هذا المثال، يحتفظ الصف *ToeButton* بمؤشر إلى الصف الأب الذي يجب أن يكون من النمط *ToeDialog*. وهو ما يؤدي إلى إنشاء ارتباط قوي لأنه لا يمكن استخدام عنصر *ToeButton* إلا مع *ToeDialog*، وهذا يحل الكثير من المشاكل.

أما الطريقة *paint()* فتساعد على إنشاء الرسوم، وذلك برسم مربع حول الزر، ورسم خطوط "x" أو "o". وكما ترى هناك الكثير من الحسابات المملة، لكنها ضرورية. ويتم النقاط نقرة الفأرة باستخدام الطريقة *mouseDown()*، حيث تقوم أولاً بالتحقق من وجود أي شيء مكتوب على الزر. فإذا لم تجد شيئاً ما فيتم الاستفسار في النافذة الأم لمعرفة من قام بإدارة الزر ولتحديد حالة الزر. يقوم بعدها هذا الزر بإجراء عملية القلب بين "x" و "o".

أما باني الصف *ToeDialog* فهو بسيط جداً، حيث يقوم بإضافة ماتحتاجه من الأزرار إلى التخطيط *GridLayout*، ثم يقوم بتغيير حجمها إلى 50 نقطة ضوئية في جميع جوانب الأزرار. لاحظ أيضاً بأن *handleEvent()* تقوم فقط باستدعاء الطريقة *dispose()* عند الحدث *WINDOW_DESTROY* مما يبقي على التطبيق قيد الاستخدام.

مكتبة AWT ضمن الإصدار 1.1 Java...

تم في الإصدار 1.1 Java إجراء تغييرا كاملا على نموذج الحدث الذي استخدم في الإصدار السابق والذي كان يعاني من نقاط ضعف كثيرة. فلقد أصبح النموذج الجديد غرضي التوجه بحيث أصبح يتعامل مع عناصر مصادر *sources* وعناصر مستمعين *listeners* من الأحداث. كما أصبح تمثيل الأحداث على شكل تسلسل هرمي من الصفوف بدلا من صف وحيد، وبإمكانك إنشاء نمط حدث خاص بك. كما قامت المكتبة الجديدة بإجراء التغييرات على أسماء الطرق، فبدلا من *setSize()* أصبح لدينا *resize()* وهو ما سيكون له معنى أفضل عندما نتعامل مع *Java Beans*.

طبعاً استمرت المكتبة الجديدة بدعم مكتبة AWT القديمة لضمان التوافقية مع البرامج الموجودة مسبقاً.

نموذج الحدث الجديد New Event Model ...

يمكن لكل مكون في هذا النموذج قَدْح حدث. ويتم تمثيل كل نمط حدث بصف منفصل. وعندما يتم قَدْح حدث، يقوم مستمع *listener* أو أكثر بتلقي هذا الحدث. لذلك يمكن أن يكون مصدر الحدث والمكان الذي سيعالج فيه منفصلين. وكل مستمع حدث *event listener* عبارة عن عنصر يقوم بتنفيذ نمط مستمع خاص *interface*. لذلك كل ماعليك عمله كمبرمج هو إنشاء عنصر مستمع وتسجيله مع المكون الذي قام بقَدْح هذا الحدث. وتتم عملية التسجيل باستدعاء الطريقة *addXXXListener()* في مكون الحدث الذي تم قَدْحه، حيث *XXX* يمثل نمط الحدث المستمع.

يوضح المثال التالي كيفية استخدام نموذج الحدث الجديد لالتقاط حدث الضغط على زر ومقارنته مع النموذج القديم :

```
//: Button2New.java
// Capturing button presses
import java.awt.*;
import java.awt.event.*; // Must add this
import java.applet.*;
public class Button2New extends Applet {
    Button
        b1 = new Button("Button 1"),
        b2 = new Button("Button 2");
    public void init() {
        b1.addActionListener(new B1());
        b2.addActionListener(new B2());
        add(b1);
        add(b2);
    }
    class B1 implements ActionListener {
        public void actionPerformed(ActionEvent e)
        {
            getAppletContext().showStatus("Button
            1");
        }
    }
    class B2 implements ActionListener {
        public void actionPerformed(ActionEvent e)
        {
            getAppletContext().showStatus("Button
            2");
        }
    }
    /* The old way:
    public boolean action(Event evt, Object arg)
    {
        if(evt.target.equals(b1))
            getAppletContext().showStatus("Button 1");
        else if(evt.target.equals(b2))
            getAppletContext().showStatus("Button 2");
        // Let the base class handle it:
        else
```



```

return super.action(evt, arg);
return true; // We've handled it here
}
*/
} ///:~

```

تستطيع هنا المقارنة بين النموذجين، فالترميز القديم ترك كتعليق. لاحظ أن التغيير الوحيد الذي تم إجراؤه على الطريقة `init()` هو إضافة السطرين:

```

b1.addActionListener(new B1());
b2.addActionListener(new B2());

```

فالطريقة `addActionListener()` تخبر الزر عن العنصر الذي ستقوم بتنفيذه عند ضغط هذا الزر. أما الصفوف `B1` و `B2` فهي عبارة عن صفوف داخلية تقوم بتنفيذ الواجهة `interface ActionListener` والتي تحتوي على الطريقة الوحيدة `.actionPerformed()`

وإن بساطة الطريقة `actionPerformed()` تعد أحد الأمور الهامة فيها مقارنة مع الطريقة القديمة `action()` والتي يتوجب عليك فيها معرفة ما الذي حدث وفيما إذا حدث بشكل سليم، والتحقق من إصدار الصف الأساسي لهذه الطريقة، وإرجاع قيمة لتحديد فيما إذا تمت معالجتها.

الحدث وأنماط المستمع *Event and listener*

...types

كما ذكرنا سابقاً، فلقد تم تغيير جميع مكونات المكتبة `AWT` لكي تتضمن الطرق `removeXXXListener()` و `addXXXListener()` بحيث تتمكن من إضافة أو حذف أنماط المستمعين المناسبة من أي مكون. ستلاحظ أيضاً بأن `XXX` تمثل في كل حالة وسيط الطريقة، خذ كمثال الطريقة `.addMouseListener(MouseListener ml)`

والجدول التالي يحتوي على الأحداث المرتبطة والمستمعين والطرق والمكونات التي تدعم هذه الأحداث الخاصة:

المكونات التي تدعم هذا الحدث	الحدث وواجهة المستمع وطرق الإضافة والحذف
<i>Button, List, TextField, MenuItem, and its derivatives including CheckboxMenuItem, Menu, and PopupMenu</i>	<i>ActionEvent ActionListener addActionListener() removeActionListener()</i>
<i>Scrollbar Anything you create that implements the Adjustable interface</i>	<i>AdjustmentEvent AdjustmentListener addAdjustmentListener() removeAdjustmentListener()</i>
<i>Component and its derivatives, including Button, Canvas, Checkbox, Choice, Container, Panel, Applet, ScrollPane, Window, Dialog, FileDialog, Frame, Label, List, Scrollbar, TextArea, and TextField</i>	<i>ComponentEvent ComponentListener addComponentListener() removeComponentListener()</i>
<i>Container and its derivatives, including Panel, Applet, ScrollPane, Window, Dialog, FileDialog, and Frame</i>	<i>ContainerEvent ContainerListener addContainerListener() removeContainerListener()</i>
<i>Component and its derivatives, including Button, Canvas, Checkbox, Choice, Container, Panel, Applet, ScrollPane, Window, Dialog, FileDialog, Frame Label, List, Scrollbar,</i>	<i>FocusEvent FocusListener addFocusListener() removeFocusListener()</i>



<i>TextArea, and TextField</i>	
<i>Component and its derivatives, including Button, Canvas, Checkbox, Choice, Container, Panel, Applet, ScrollPane, Window, Dialog, FileDialog, Frame, Label, List, Scrollbar, TextArea, and TextField</i>	<i>KeyEvent</i> <i>KeyListener</i> <i>addKeyListener()</i> <i>removeKeyListener()</i>
<i>Component and its derivatives, including Button, Canvas, Checkbox, Choice, Container, Panel, Applet, ScrollPane, Window, Dialog, FileDialog, Frame, Label, List, Scrollbar, TextArea, and TextField</i>	<i>MouseEvent (for both clicks and motion)</i> <i>MouseListener</i> <i>addMouseListener()</i> <i>removeMouseListener()</i>
<i>Component and its derivatives, including Button, Canvas, Checkbox, Choice, Container, Panel, Applet, ScrollPane, Window, Dialog, FileDialog, Frame, Label, List, Scrollbar, TextArea, and TextField</i>	<i>MouseEvent (for both clicks and motion)</i> <i>MouseMotionListener</i> <i>addMouseMotionListener()</i> <i>removeMouseMotionListener()</i> <i>)</i>
<i>Window and its derivatives, including Dialog, FileDialog, and Frame</i>	<i>WindowEvent</i> <i>WindowListener</i> <i>addWindowListener()</i> <i>removeWindowListener()</i>
<i>Checkbox, CheckboxMenuItem,</i>	<i>ItemEvent</i> <i>ItemListener</i>

<i>Choice, List, and anything that implements the ItemSelectable interface</i>	<i>addItemListener() removeItemListener()</i>
<i>Anything derived from TextComponent, including TextArea and TextField</i>	<i>TextEvent TextListener addTextListener() removeTextListener()</i>

تستطيع ملاحظة أن كل نمط مكون يدعم أنماط محددة من الأحداث. ومن المفيد معرفة الأحداث المدعومة من قبل كل مكون و هي موضحة في الجدول التالي:

نمط المكون	الأحداث المدعومة من قبل هذا المكون
<i>Adjustable</i>	<i>AdjustmentEvent</i>
<i>Applet</i>	<i>ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>
<i>Button</i>	<i>ActionEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>
<i>Canvas</i>	<i>FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>
<i>Checkbox</i>	<i>ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>
<i>CheckboxMenuItem</i>	<i>ActionEvent, ItemEvent</i>
<i>Choice</i>	<i>ItemEvent, FocusEvent,</i>



<i>KeyEvent,</i> <i>MouseEvent,</i> <i>ComponentEvent</i>	
<i>FocusEvent, KeyEvent,</i> <i>MouseEvent,</i> <i>ComponentEvent</i>	<i>Component</i>
<i>ContainerEvent, FocusEvent,</i> <i>KeyEvent, MouseEvent,</i> <i>ComponentEvent</i>	<i>Container</i>
<i>ContainerEvent,</i> <i>WindowEvent,</i> <i>FocusEvent, KeyEvent,</i> <i>MouseEvent,</i> <i>ComponentEvent</i>	<i>Dialog</i>
<i>ContainerEvent,</i> <i>WindowEvent, FocusEvent,</i> <i>KeyEvent, MouseEvent,</i> <i>ComponentEvent</i>	<i>FileDialog</i>
<i>ContainerEvent,</i> <i>WindowEvent,</i> <i>FocusEvent, KeyEvent,</i> <i>MouseEvent,</i> <i>ComponentEvent</i>	<i>Frame</i>
<i>FocusEvent, KeyEvent,</i> <i>MouseEvent,</i> <i>ComponentEvent</i>	<i>Label</i>
<i>ActionEvent, FocusEvent,</i> <i>KeyEvent,</i>	<i>List</i>

<i>MouseEvent, ItemEvent, ComponentEvent</i>	
<i>ActionEvent</i>	<i>Menu</i>
<i>ActionEvent</i>	<i>MenuItem</i>
<i>ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>	<i>Panel</i>
<i>ActionEvent</i>	<i>PopupMenu</i>
<i>AdjustmentEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>	<i>Scrollbar</i>
<i>ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>	<i>ScrollPane</i>
<i>TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>	<i>TextArea</i>
<i>TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>	<i>TextComponent</i>
<i>ActionEvent, TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent</i>	<i>TextField</i>
<i>ContainerEvent,</i>	<i>Window</i>



<i>WindowEvent,</i> <i>FocusEvent, KeyEvent,</i> <i>MouseEvent,</i> <i>ComponentEvent</i>	
--	--

وعندما تعرف الأحداث التي يدعمها مكون ما، فلن تحتاج للبحث عن أي شيء للتفاعل مع هذه الأحداث، كل ما عليك عمله هو:

١. أخذ اسم صف الحدث وحذف الكلمة *Event*، ثم إضافة الكلمة *Listener* "ستحصل على واجهة المستمع التي تحتاجها للتنفيذ في صفك الداخلي".
 ٢. قم بتنفيذ الواجهة السابقة واكتب طرق الأحداث التي ترغب بالتقاطها. فقد تحتاج مثلا للبحث عن حركات الفأرة، لذلك اكتب الترميز الموافق للطريقة *mouseMoved()* الخاصة بالواجهة *MouseMotionListener*.
 ٣. أنشئ عنصر صف المستمع *listener class*. احفظه مع المكون وكذلك مع الطريقة الناتجة بإضافة *"add"* في بداية اسم المستمع. مثلا *addMouseMotionListener()*.
- لإنهاء مايتوجب عليك معرفته، إليك واجهات المستمع *listener* :
:interfaces

طرق الواجهة	واجهة المستمع
<i>actionPerformed(ActionEvent)</i>	<i>ActionListener</i>
<i>adjustmentValueChanged(AdjustmentEvent)</i>	<i>AdjustmentListener</i>
<i>componentHidden(ComponentEvent)</i> <i>componentShown(ComponentEvent)</i> <i>componentMoved(ComponentEvent)</i> <i>componentResized(ComponentEvent)</i>	<i>ComponentListener</i> <i>ComponentAdapter</i>
<i>componentAdded(ContainerEvent)</i> <i>componentRemoved(ContainerEvent)</i>	<i>ContainerListener</i> <i>ContainerAdapter</i>

<i>focusGained(FocusEvent)</i>	<i>FocusListener</i>
<i>focusLost(FocusEvent)</i>	<i>FocusAdapter</i>
<i>keyPressed(KeyEvent)</i>	<i>KeyListener</i>
<i>keyReleased(KeyEvent)</i>	<i>KeyAdapter</i>
<i>keyTyped(KeyEvent)</i>	
<i>mouseClicked(MouseEvent)</i>	<i>MouseListener</i>
<i>mouseEntered(MouseEvent)</i>	<i>MouseAdapter</i>
<i>mouseExited(MouseEvent)</i>	
<i>mousePressed(MouseEvent)</i>	
<i>mouseReleased(MouseEvent)</i>	
<i>mouseDragged(MouseEvent)</i>	<i>MouseMotionListener</i>
<i>mouseMoved(MouseEvent)</i>	<i>MouseMotionAdapter</i>
<i>windowOpened(WindowEvent)</i>	<i>WindowListener</i>
<i>windowClosing(WindowEvent)</i>	<i>WindowAdapter</i>
<i>windowClosed(WindowEvent)</i>	
<i>windowActivated(WindowEvent)</i>	
<i>windowDeactivated(WindowEvent)</i>	
<i>windowIconified(WindowEvent)</i>	
<i>windowDeiconified(WindowEvent)</i>	
<i>itemStateChanged(ItemEvent)</i>	<i>ItemListener</i>
<i>textValueChanged(TextEvent)</i>	<i>TextListener</i>

كتطبيق على ذلك سنقوم بإعادة كتابة المثال الخاص بالحقول النصية *Text Fields* وذلك باستخدام مكتبة *AWT* الجديدة:

```
//: TextNew.java
// Text fields with Java 1.1 events
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```



```

public class TextNew extends Applet {
    Button
        b1 = new Button("Get Text"),
        b2 = new Button("Set Text");
    TextField
        t1 = new TextField(30),
        t2 = new TextField(30),
        t3 = new TextField(30);
    String s = new String();
    public void init() {
        b1.addActionListener(new B1());
        b2.addActionListener(new B2());
        t1.addTextListener(new T1());
        t1.addActionListener(new T1A());
        t1.addKeyListener(new T1K());
        add(b1);
        add(b2);
        add(t1);
        add(t2);
        add(t3);
    }
    class T1 implements TextListener {
        public void textValueChanged(TextEvent e) {
            t2.setText(t1.getText());
        }
    }
    class T1A implements ActionListener {
        private int count = 0;
        public void actionPerformed(ActionEvent e)
        {
            t3.setText("t1 Action Event " + count++);
        }
    }
    class T1K extends KeyAdapter {
        public void keyTyped(KeyEvent e) {
            String ts = t1.getText();
            if(e.getKeyChar() ==
                KeyEvent.VK_BACK_SPACE) {

```

```
// Ensure it's not empty:
if( ts.length() > 0) {
    ts = ts.substring(0, ts.length() -
1);
    t1.setText(ts);
}
else
    t1.setText(
        t1.getText() +
        Character.toUpperCase(
            e.getKeyChar()));
    t1.setCaretPosition(
        t1.getText().length());
// Stop regular character from appearing:
e.consume();
}
}
class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e)
    {
        s = t1.getSelectedText();
        if(s.length() == 0) s = t1.getText();
        t1.setEditable(true);
    }
}
class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e)
    {
        t1.setText("Inserted by Button 2: " + s);
        t1.setEditable(false);
    }
}
public static void main(String[] args) {
    TextNew applet = new TextNew();
    Frame aFrame = new Frame("TextNew");
    aFrame.addWindowListener(
        new WindowAdapter() {
```



```

public void windowClosing(WindowEvent
e) {
    System.exit(0);
}
});
aFrame.add(applet, BorderLayout.CENTER);
aFrame.setSize(300,200);
applet.init();
applet.start();
aFrame.setVisible(true);
}
} ///:~

```

لاحظ هنا بأنه قد تم تضمين العنصر `TextField t3` عند قـدح مستمع الفعل `listener action` الخاص بالعنصر `TextField t1`. وسترى بأنه سيتم قـدح هذا المستمع عند ضغط مفتاح `enter` فقط.

ويمتلك العنصر `TextField t1` عدة مستمعين متصلين به. فالمستمع `T1` يقوم بنسخ النص من `t1` إلى `t2`، أما المستمع `T2K` فيقوم بتحويل جميع الحارف إلى أحرف كبيرة. ستلاحظ أيضا بأن هذين المستمعين يعملان سوياً.

يبرهن هذا المثال على فائدة تصميم الصفوف الداخلية `inner classes`. لاحظ هنا بأنه في الصف الداخلي التالي:

```

class T1 implements TextListener {
    public void textValueChanged(TextEvent e) {
        t2.setText(t1.getText());
    }
}

```

فإن `t1` و `t2` لا ينتميان إلى `T1`، مما يساعد على الوصول إليهما دون أية عراقيل. والسبب في ذلك هو أنه يمكن لأي عنصر في صف داخلي التقاط مؤشر إلى الصف الخارجي الذي قام بإنشائه تلقائياً.

وأهم ما يميز مكتبة `AWT` الجديدة هو المرونة `flexibility`. فقد كنت في النموذج القديم مجبراً على إجراء الترميز القاسي لسلوك برنامج، أما في النموذج الجديد فلا تحتاج

لإضافة أو حذف سلوك حدث إلا لاستدعاء طريقة وحيدة. والمثال التالي يوضح ماسبق أن ذكرناه:

```
//: DynamicEvents.java
// The new Java 1.1 event model allows you to
// change event behavior dynamically. Also
// demonstrates multiple actions for an event.
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class DynamicEvents extends Frame {
    Vector v = new Vector();
    int i = 0;
    Button
        b1 = new Button("Button 1"),
        b2 = new Button("Button 2");
    public DynamicEvents() {
        setLayout(new FlowLayout());
        b1.addActionListener(new B());
        b1.addActionListener(new B1());
        b2.addActionListener(new B());
        b2.addActionListener(new B2());
        add(b1);
        add(b2);
    }
    class B implements ActionListener {
        public void actionPerformed(ActionEvent e)
        {
            System.out.println("A button was pressed");
        }
    }
    class CountListener implements ActionListener
    {
        int index;
        public CountListener(int i) { index = i; }
        public void actionPerformed(ActionEvent e)
        {
            System.out.println(
```



```

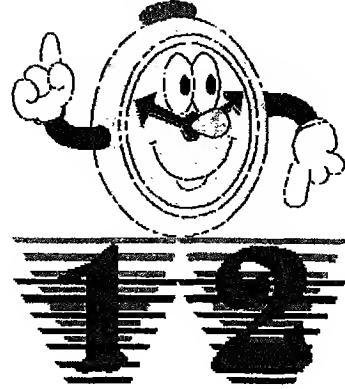
        "Counted Listener " + index);
    }
}
class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("Button 1 pressed");
        ActionListener a = new
        CountListener(i++);
        v.addElement(a);
        b2.addActionListener(a);
    }
}
class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("Button 2 pressed");
        int end = v.size() -1;
        if(end >= 0) {
            b2.removeActionListener(
                (ActionListener)v.elementAt(end));
            v.removeElementAt(end);
        }
    }
}
public static void main(String[] args) {
    Frame f = new DynamicEvents();
    f.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    f.setSize(300,200);
    f.show();
}
} ///:~

```

الأمور الهامة الجديدة في هذا المثال تتلخص بالنقاط التالية:

١. يوجد أكثر من مستمع متصل بكل *Button*.
٢. أثناء تنفيذ البرنامج، تتم عملية إضافة أو حذف المستمعين من الزر *Button b2* بشكل ديناميكي.





بدأت البرمجة المرئية *Visual Programming* بالانتشار بنجاح عند إصدار شركة مايكروسوفت النسخة الأولى *Visual Basic*. أصدرت بعدها شركة بورلاند نسختها من *Borland Delphi* معلنة بدء الجيل الثاني من تصميم البرامج المرئية (وكانت الـ *Java Beans*).

وباستخدام أدوات البرمجة هذه، يتم تمثيل المكونات بشكل مرئي. وغالباً ما يكون التمثيل المرئي للمكونات مشابه تماماً لما سنراه عند تنفيذ البرنامج. والإجراءات الأساسية المتبعة في هذا النوع من البرمجة تعتمد على سحب مكون ما من لوحة جانبيّة وإفلاته على النموذج، ويقوم باني التطبيق *application builder* بتوليد الترميز الموافق لإنشاء هذا المكون.

طبعاً عملية سحب المكونات وإفلاتها على نموذجك لن تكون كافية لإتمام البرنامج، بل يتوجب عليك تغيير مواصفات هذه المكونات في أغلب الأحيان، كاللون والنص الذي يمكن أن تحتويه وغير ذلك. وتعرف هذه المواصفات في البرمجة المرئية بالخصائص *properties*.

إضافة إلى ذلك فإن أي عنصر في البرمجة المرئية ليس مجموعة من الخصائص فقط، بل هو مجموعة من السلوك *behaviors* أيضاً، ويتم تحديدها عند التصميم وتعرف بالأحداث *Events*.

حبيبات جافا Java Beans ...

أتاحت لغة جافا إمكانية إنشاء مرئية عن طريق ما يسمى بحبيبات جافا *Java Beans*. وكل حبيبة *Bean* عبارة عن صف، ولن تكون بحاجة إلى كتابة أي ترميز من أجل إنشائه. الشيء الوحيد الذي ستقوم به هو إجراء تغيير بسيط على طريقة تسمية الطرق. لأن اسم الطريقة هو الذي يخبر أداة باني التطبيق *application builder* *tool*، فيما إذا كانت هذه الطريقة خاصة *property* أو حدث *event* أو طريقة عادية *ordinary method*.

ويتم استخدام الطريقة الاصطلاحية التالية لاختيار التسميات:

١. يتم إنشاء طريقتين: *getXxx()* و *setXxx()*، لكل خاصية بالاسم *xxx*.
٢. يمكن استخدام الطريقة المحددة في النقطة السابقة، من أجل الخاصية المنطقية، يمكنك أيضاً استخدام *"is"* بدلاً من *"get"*.

٣. الطرق العادية الخاصة بكل حبيبة *Bean* لاتأخذ نفس طريقة التسميات التي ذكرناها، لكنها عامة *public*.

٤. بالنسبة للأحداث، يمكنك استخدام طريقة المستمع *listener*. وهي تشبه تماماً ما ذكرناه سابقاً، حيث يمكن استخدام الطريقة *addFooBarListener(FooBarListener)* و *removeFooBarListener(FooBarListener)* لمعالجة الحدث *FooBarEvent*. وفي أغلب الأحيان يمكن للأحداث المبنية مسبقاً *built-in* *events* و المستمعين *listeners* تلبية احتياجاتك، لكن بإمكانك إنشاء أحداثك وواجهات المستمع الخاصة بك.

تجيبك النقطة الأولى عن الكثير من التساؤلات التي تم طرحها عند الانتقال من الإصدار *Java 1.0* إلى الإصدار *Java 1.1* خاصة على تسميات الطرق. وكما تلاحظ الآن بأن أغلب التغييرات تمت من أجل التوافق مع إضافة *"get"* و *"set"* للتسميات الاصطلاحية، وكل ذلك لتحويل مكون ما إلى حبيبة *Bean*.
يمكننا الآن استخدام النصائح السابقة لإنشاء حبيبة *Bean* بسيطة:

```
//: Frog.java
// A trivial Java Bean
package frogbean;
import java.awt.*;
import java.awt.event.*;
class Spots {}
public class Frog {
    private int jumps;
    private Color color;
    private Spots spots;
    private boolean jmp;
    public int getJumps() { return jumps; }
    public void setJumps(int newJumps) {
        jumps = newJumps;
    }
    public Color getColor() { return color; }
    public void setColor(Color newColor) {
```

```

color = newColor;
}
public Spots getSpots() { return spots; }
public void setSpots(Spots newSpots) {
    spots = newSpots;
}
public boolean isJumper() { return jmpr; }
public void setJumper(boolean j) { jmpr = j;
}
public void addActionListener(
    ActionListener l) {
    //...
}
public void removeActionListener(
    ActionListener l) {
    // ...
}
public void addKeyListener(KeyListener l) {
    // ...
}
public void removeKeyListener(KeyListener l)
{
    // ...
}
// An "ordinary" public method:
public void croak() {
    System.out.println("Ribbet!");
}
} ///:~

```

لاحظ أولاً بأن كل حبيبة Bean عبارة عن صف `class`. وعادة تكون جميع الحقول فيها خاصة `private`، ولا يمكن الوصول إليها إلا من خلال الطرق. وباتباع التسمية الاصطلاحية فإن الخصائص هي: `jumps, color, spots, jumper`. لاحظ بأن اسم المحدد الداخلي `internal identifier` هو نفس اسم الخاصية في الحالات الثلاث الأولى، أما في `jumper` فتلاحظ بأن اسم الخاصية هنا لا يجبرك على استخدام تسميات معينة للمتحوّلات الداخلية.

أما الأحداث التي يمكن لـ *Bean* معالجتها فهي *ActionEvent* و *KeyEvent*.

أخيراً يمكنك ملاحظة بقاء الطريقة الاعتيادية (*croak*) كجزء من الحبيبة *Bean* لأنها ببساطة طريقة عامة *public*، ولاتتوافق مع أي تسمية اصطلاحية.

الحصول على *BeanInfo* باستخدام *Introspector*...

تعتبر عملية سحب حبيبة *Bean* من لوحة وإفلاتها على نموذج من العمليات الحرجة. ويجب أن تتمكن أداة بناء التطبيق *application builder* من إنشاء حبيبة *Bean*، ومن ثم جلب جميع المعلومات الضرورية لإنشاء صفحة الخصائص *property sheet* ومؤشرات الحدث *event handlers* (دون النفاذ إلى الترميز المصدر للحبيبة *Bean*).

وتسمح خاصية الانعكاس *reflection*، التي أتت مع الإصدار *Java 1.1*، باكتشاف جميع طرق صف مجهول. وهو ما يساعد على حل مشكلة الحبيبة *Bean* دون أن يتطلب ذلك منك استخدام آلية كلمات مفاتيح خاصة، كالتي يتم استخدامها في بقية لغات البرمجة المرئية. وفي الواقع فإن أحد أهم الأسباب الرئيسية لإضافة خاصية الانعكاس إلى الإصدار *Java 1.1* هو من أجل دعم الحبيبة *Beans*. لذلك يمكنك أن تتوقع بأن باني التطبيق يقوم بعكس عملية إنشاء حبيبة *Bean*، وكشف الطرق من خلال ذلك من أجل إيجاد خصائص وأحداث هذه الحبيبة *Bean*.

ويحتاج مصممو جافا إلى إعطاء واجهة قياسية للحبيبة *Bean* كي يتمكن أيًا كان من استخدامها بشكل أبسط. ويتم ذلك باستخدام الصف *Introspector* والذي يحتوي على الطريقة الهامة (*static getBeanInfo*). قم فقط بتمرير مؤشر هذا الصف إلى الطريقة السابقة، فنقوم باستجواب الصف وإرجاع عنصر *BeanInfo* يحتوي على معلومات عن خصائص وطرق وأحداث هذه الحبيبة *Bean*.

سنقوم في المثال التالي بتوضيح كيفية استخدام الصف *Introspector* لإظهار معلومات عن الحبيبة *Bean*:

```
//: BeanDumper.java
// A method to introspect a Bean
import java.beans.*;
import java.lang.reflect.*;
public class BeanDumper {
    public static void dump(Class bean){
        BeanInfo bi = null;
        try {
            bi = Introspector.getBeanInfo(
                bean, java.lang.Object.class);
        } catch(IntrospectionException ex) {
            System.out.println("Couldn't introspect "
                +bean.getName());
            System.exit(1);
        }
        PropertyDescriptor[] properties =
            bi.getPropertyDescriptors();
        for(int i = 0; i < properties.length; i++)
        {
            Class p =
                properties[i].getPropertyType();
            System.out.println(
                "Property type:\n " + p.getName());
            System.out.println(
                "Property name:\n " +
                    properties[i].getName());
            Method readMethod =
                properties[i].getReadMethod();
            if(readMethod != null)
                System.out.println(
                    "Read method:\n " +
                        readMethod.toString());
            Method writeMethod =
                properties[i].getWriteMethod();
            if(writeMethod != null)
```

```

System.out.println(
    "Write method:\n " +
    writeMethod.toString());
System.out.println("=====
==");
}
System.out.println("Public methods:");
MethodDescriptor[] methods =
    bi.getMethodDescriptors();
for(int i = 0; i < methods.length; i++)
    System.out.println(
        methods[i].getMethod().toString());
System.out.println("=====
");
System.out.println("Event support:");
EventSetDescriptor[] events =
    bi.getEventSetDescriptors();
for(int i = 0; i < events.length; i++) {
    System.out.println("Listener type:\n " +
        events[i].getListenerType().getName());
    Method[] lm =
        events[i].getListenerMethods();
    for(int j = 0; j < lm.length; j++)
        System.out.println(
            "Listener method:\n " +
            lm[j].getName());
    MethodDescriptor[] lmd =
        events[i].getListenerMethodDescriptors();
    for(int j = 0; j < lmd.length; j++)
        System.out.println(
            "Method descriptor:\n " +
            lmd[j].getMethod().toString());
    Method addListener =
        events[i].getAddListenerMethod();
    System.out.println(
        "Add Listener Method:\n " +
        addListener.toString());
    Method removeListener =

```

```

events[i].getRemoveListenerMethod();
System.out.println(
    "Remove Listener Method:\n " +
    removeListener.toString());
System.out.println("====="
);
}
}
// Dump the class of your choice:
public static void main(String[] args) {
    if(args.length < 1) {
        System.err.println("usage: \n" +
            "BeanDumper fully.qualified.class");
        System.exit(0);
    }
    Class c = null;
    try {
        c = Class.forName(args[0]);
    } catch(ClassNotFoundException ex) {
        System.err.println(
            "Couldn't find " + args[0]);
        System.exit(0);
    }
    dump(c);
}
} ///:~

```

كما تلاحظ فإن الطريقة `BeanDumper.dump()` تقوم بإنجاز أغلب الأعمال المطلوبة، فهي تحاول أولاً إنشاء عنصر `BeanInfo`، وعندما تنجح فإنها تقوم باستدعاء طرق الصف `BeanInfo` التي تساعد على إعطاء معلومات عن خصائص وطرق وأحداث هذا الـ `Bean`. أما في الطريقة `Introspector.getBeanInfo()` فتلاحظ وجود وسيط ثانٍ يخبر الصف `Introspector` عن المكان الذي يجب أن نتوقف فيه ضمن هرمية التوريث `.inheritance hierarchy`.

أما الطريقة `getPropertyDescriptors()` فتقوم بإرجاع مصفوفة خصائص `PropertyDescriptor`، وفي كل خاصية منها يمكن استدعاء الطريقة `getPropertyType()` لإيجاد الصف الذي تم تمريره إلى طرق الخاصية. ويمكنك الحصول على اسم كل خاصية باستخدام `getName()`، وكذلك طريقة القراءة `getReadMethod()` وطريقة الكتابة `getWriteMethod()`. وتقوم الطريقتان السابقتان بإرجاع عنصر `Method` يمكن استخدامه لتنفيذ الطريقة الموافقة. أما بالنسبة للطرق العامة، فإن الطريقة `getMethodDescriptor()` تقوم بإرجاع مصفوفة العناصر `MethodDescriptor`. ومن هذه العناصر يمكنك الحصول على عنصر `Object` الموافق وطباعة اسمه. وبالنسبة للأحداث، تقوم الطريقة `getEventSetDescriptors()` بإرجاع مصفوفة العناصر `EventSetDescriptors`. ويمكن استعلام أيًا منها لإيجاد الصف الخاص بالمستمع `listener`، وطرق الصف الخاص بهذا المستمع، وطرق إضافة وحذف المستمع؛ حيث يتكفل البرنامج `BeanDumper` بطباعة جميع هذه المعلومات.

الآن إذا قمت بتشغيل البرنامج `BeanDumper` على الصف `Frog` وفق الشكل:

```
java BeanDumper frogbean.Frog
```

سيكون الخرج (بعد حذف بعض التفاصيل غير الضرورية) على الشكل:

```
class name: Frog
Property type:
  Color
Property name:
  color
Read method:
public Color getColor()
Write method:
public void setColor(Color)
=====
Property type:
  Spots
Property name:
```

سلسلة الرضا للمعلومات

```

    spots
Read method:
    public Spots getSpots()
Write method:
    public void setSpots(Spots)
=====
Property type:
    boolean
Property name:
    jumper
Read method:
    public boolean isJumper()
Write method:
    public void setJumper(boolean)
=====
Property type:
    int
Property name:
    jumps
Read method:
    public int getJumps()
Write method:
    public void setJumps(int)
=====
Public methods:
    public void setJumps(int)
    public void croak()
    public void
    removeActionListener(ActionListener)
    public void addActionListener(ActionListener)
    public int getJumps()
    public void setColor(Color)
    public void setSpots(Spots)
    public void setJumper(boolean)
    public boolean isJumper()
    public void addKeyListener(KeyListener)
    public Color getColor()
    public void removeKeyListener(KeyListener)

```

```
public Spots getSpots()
```

```
=====
```

```
Event support:
```

```
Listener type:
```

```
KeyListener
```

```
Listener method:
```

```
keyTyped
```

```
Listener method:
```

```
keyPressed
```

```
Listener method:
```

```
keyReleased
```

```
Method descriptor:
```

```
public void keyTyped(KeyEvent)
```

```
Method descriptor:
```

```
public void keyPressed(KeyEvent)
```

```
Method descriptor:
```

```
public void keyReleased(KeyEvent)
```

```
Add Listener Method:
```

```
public void addKeyListener(KeyListener)
```

```
Remove Listener Method:
```

```
public void removeKeyListener(KeyListener)
```

```
=====
```

```
Listener type:
```

```
ActionListener
```

```
Listener method:
```

```
actionPerformed
```

```
Method descriptor:
```

```
public void actionPerformed(ActionEvent)
```

```
Add Listener Method:
```

```
public void addActionListener(ActionListener)
```

```
Remove Listener Method:
```

```
public void
```

```
removeActionListener(ActionListener)
```

```
=====
```

سأعطيك الآن مثلاً مسلياً أكثر...

سنقوم في المثال التالي بإنشاء كنافا نقوم برسم دائرة صغيرة حول مؤشر الفأرة عندما تتحرك، وعندما نقوم بضغط الفأرة تظهر الكلمة "Bang!" في وسط الشاشة. الخصائص التي بإمكانك تغييرها هي حجم الدائرة ولونها، إضافةً إلى حجم وخصائص النص الذي سيظهر عند ضغط الفأرة.

```
//: BangBean.java
// A graphical Bean
package bangbean;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
public class BangBean extends Canvas
    implements Serializable {
    protected int xm, ym;
    protected int cSize = 20; // Circle size
    protected String text = "Bang!";
    protected int fontSize = 48;
    protected Color tColor = Color.red;
    protected ActionListener actionListener;
    public BangBean() {
        addMouseListener(new ML());
        addMouseMotionListener(new MML());
    }
    public int getCircleSize() { return cSize; }
    public void setCircleSize(int newSize) {
        cSize = newSize;
    }
    public String getBangText() { return text; }
    public void setBangText(String newText) {
        text = newText;
    }
    public int getFontSize() { return fontSize; }
```

```

public void setFontSize(int newSize) {
    fontSize = newSize;
}
public Color getTextColor() { return tColor;
}
public void setTextColor(Color newColor) {
    tColor = newColor;
}
public void paint(Graphics g) {
    g.setColor(Color.black);
    g.drawOval(xm - cSize/2, ym - cSize/2,
               cSize, cSize);
}
// This is a unicast listener, which is
// the simplest form of listener management:
public void addActionListener (
    ActionListener l)
    throws TooManyListenersException {
    if(actionListener != null)
        throw new TooManyListenersException();
    actionListener = l;
}
public void removeActionListener(
    ActionListener l) {
    actionListener = null;
}
class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font(
                "TimesRoman", Font.BOLD, fontSize));
        int width =
            g.getFontMetrics().stringWidth(text);
        g.drawString(text,
            (getSize().width - width) /2,
            getSize().height/2);
    }
}

```

```

g.dispose();
// Call the listener's method:
if(actionListener != null)
    actionPerformed(
        new ActionEvent(BangBean.this,
            ActionEvent.ACTION_PERFORMED,
            null));
}
}
class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}
public Dimension getPreferredSize() {
    return new Dimension(200, 200);
}
// Testing the BangBean:
public static void main(String[] args) {
    BangBean bb = new BangBean();
    try {
        bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {}
    Frame aFrame = new Frame("BangBean Test");
    aFrame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent
                e) {
                System.exit(0);
            }
        });
    aFrame.add(bb, BorderLayout.CENTER);
    aFrame.setSize(300,300);
    aFrame.setVisible(true);
}
// During testing, send action information

```

```
// to the console:
static class BBL implements ActionListener
{
    public void actionPerformed(ActionEvent
    e) {
        System.out.println("BangBean action");
    }
}
} ///:~
```

الأمر الهام الذي يجب ملاحظته هو أن الصف *BangBean* يقوم بتنفيذ الواجهة *Serializable*. هذا يعني أن بإمكان أداة باني التطبيق الحصول على جميع المعلومات من *BangBean* باستخدام التسلسلية *serialization*، وذلك بعد أن يقوم مصمم البرنامج بتحديد قيم الخصائص.

لاحظ أن بإمكان الطريقة *addActionListener()* قذف الاستثناء *TooManyListenersException*. وهو يقوم بإعلام مستمع وحيد عند حصول الحدث.

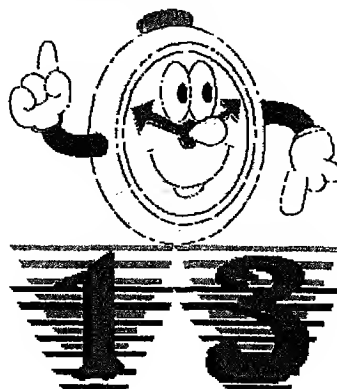
وعندما تقوم بضغط زر الفأرة، يتم وضع النص في مركز العنصر *BangBean*، ويتم استدعاء الطريقة *actionPerformed()* عندما تكون قيمة الحقل *actionListener* غير معدومة، حيث يتم إنشاء عنصر *ActionEvent* جديد. وعندما تتحرك الفأرة، يتم التقاط إحداثياتها الجديدة، وتُرسَم الكنافا من جديد. ولقد تمت إضافة *main()* للسماح لك باختبار هذا البرنامج من خلال سطر الأوامر. طبعاً عندما تكون الحبيبة *Bean* ضمن بيئة التطوير، فلن تستخدم الطريقة *main()*، لكن من المفيد استخدام *main()* في كل *Bean* تقوم بإنشائها لأنها تمكننا من إجراء اختبار سريع عليها.

تقوم الطريقة *main()* هنا بإنشاء إطار *Frame* ووضع العنصر *BangBean* ضمنه. نقوم بعدها بربط العنصر البسيط *ActionListener* مع العنصر *BangBean* لطباعة رسالة على الشاشة تدلّ على حدوث الحدث *ActionEvent*.

تحميل الحبيبات *Packaging Beans*...

قبل أن يكون باستطاعتك وضع حبيبة *Bean* ضمن مجموعة أدوات الباني المرئية، يجب أن توضع ضمن حاوية الحبيبة *Bean* القياسية، وهي عبارة عن ملف *JAR* يحتوي على جميع صفوف *Bean* إضافةً إلى ملف لائحة الحبيبة *Bean*.

قم بعد ذلك بتنفيذ برنامج *jar* في نفس مجلد ملف اللائحة *manifest file*:
`jar cfm BangBean.jar BangBean.mf bangbean`
 حيث أن *BangBean.mf* هو اسم ملف اللائحة.



لنا العناصر بتقسيم أي برنامج إلى فقرات منفصلة، فغالبا ما نحتاج إلى **تسمح** تحويل برنامج ما إلى مهام جزئية منفصلة ومستقلة. نسمي كل مهمة من هذه المهام الفرعية بالنيسب *Thread*، ونستطيع إنشاء البرامج وكأن كل نيسب يمتلك معالج *CPU* خاصا به.

عند هذه النقطة من المفيد إعطاء بعض التعاريف: فالإجراء *process* عبارة عن برنامج تنفيذي يمتلك فضاء عنوان *address space* خاصاً به. أما أنظمة التشغيل متعددة المهام *multitasking operating system* فهي عبارة عن أنظمة تشغيل قادرة على تنفيذ أكثر من إجراء (برنامج) في نفس الوقت. ويمكن لإجراء وحيد أن يمتلك عدة نياصت تنفيذية متزامنة. يمكن الحصول على الكثير من الفوائد من تعددية النياصت *Multithreading*، فيمكن مثلاً أن يرتبط جزء ما من برنامجك بحدث أو مصدر خاص، ولا تريد تجميد بقية البرنامج. لذلك يمكنك إنشاء نياصت مرتبط بالحدث أو المصدر، وتركه يعمل بشكل مستقل عن البرنامج الرئيسي.

يمكنك إنشاء واجهات مستخدم سريعة الاستجابة...

لنفترض أن لدينا برنامجاً يقوم بإجراء بعض العمليات التي تتطلب استغلال المعالج *CPU* إلى أقصى حد، وأن هذا البرنامج بدأ بتجاهل إدخالات المستخدم وأصبح عديم الاستجابة، كما في المثال التالي الذي يقوم بإظهار نتيجة تنفيذ عدّاد ببساطة:

```
//: Counter1.java
// A non-responsive user interface
package cl4;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class Counter1 extends Applet {
    private int count = 0;
    private Button
        onOff = new Button("Toggle"),
        start = new Button("Start");
    private TextField t = new TextField(10);
```

```

private boolean runFlag = true;
public void init() {
    add(t);
    start.addActionListener(new StartL());
    add(start);
    onOff.addActionListener(new OnOffL());
    add(onOff);
}
public void go() {
    while (true) {
        try {
            Thread.currentThread().sleep(100);
        } catch (InterruptedException e) {}
        if(runFlag)
            t.setText(Integer.toString(count++));
    }
}
class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        go();
    }
}
class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
    }
}
public static void main(String[] args) {
    Counter1 applet = new Counter1();
    Frame aFrame = new Frame("Counter1");
    aFrame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        }
    );
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300,200);
}

```

```

applet.init();
applet.start();
aFrame.setVisible(true);
}
} ///:~

```

في البرنامج السابق نلاحظ بأن الطريقة (*go()*) موجودة في المكان الذي يبقى فيه البرنامج مشغولاً، وتقوم بوضع القيمة الحالية للعداد *count* في *TextField t* ثم تقوم بزيادة هذا العداد. كما توجد حلقة لانتهائية ضمن هذه الطريقة وذلك عند طلب (*sleep()*) والتي يجب أن ترتبط بعنصر ينسب *Thread*. (في الواقع فإن جافا تعتمد على النياسب والتي يعمل بعضها بشكل دائم مع تطبيقاتك). لذلك حتى لو لم تستخدم النياسب من قبل وبشكل صريح، فإنك تستطيع معرفة النيسب الحالي المستخدم من قبل برنامجك باستخدام الطريقة (*Thread.currentThread()*) ثم استدعاء (*sleep()*) على هذا النيسب.

لاحظ أيضاً أن بإمكان الطريقة (*sleep()*) قذف الاستثناء *InterruptedException* على الرغم من أنه لا ينصح بقطع نيسب عن طريق قذف استثناء.

وعند ضغط الزر *Start* تقلع الطريقة (*go()*)، وقد تظن بأن هذا سيؤدي إلى تشغيل نيسب جديد، والسبب في ذلك هو أنه عندما تكون هذه الطريقة في حالة استيقاظ سيخيل إليك بأن المعالج *CPU* مشغول بمراقبة بقية الأزرار. لكن المشكلة الحقيقية هي أن (*go()*) لن ترجع أبداً لأنها علقت في دوامة لانتهائية، مما يعني بأن (*actionPerformed()*) لن ترجع هي الأخرى، وسيلق برنامجك يا صاحبي!!؟

المشكلة الأساسية هنا هي أن (*go()*) عليها الاستمرار في إنجاز أعمالها، وفي نفس الوقت عليها الرجوع حتى تتمكن (*actionPerformed()*) من إنهاء عملها، ولتتمكن واجهة المستخدم من الاستجابة لطلباته. لكن باستخدام الطريقة التقليدية (*go()*) لن تستطيع الاستمرار وإعادة التحكم في نفس الوقت ضمن باقي البرنامج. وكأننا هنا نطلب من المعالج *CPU* أن يكون في مكانين في نفس الوقت. لذلك يأتي هنا دور النياسب *Threads* حيث يستطيع عندها المعالج البحث في الجوار وإعطاء كل نيسب جزءاً من وقته. وقد يخيل لكل

نيسب دائما بأنه يمتلك المعالج لوحده، لكن في الواقع فإن المعالج جزئى وقته بين جميع النياسب.

وربما أتاك أحدهم يتبجح بأن النياسب تقلل من الفعالية والأداء !!؟

لكن باستطاعتك إفحامه لأن تعدد النياسب يفيد في تحسين تصميم البرنامج *program design* وموازنة المصادر *resources balancing* وعدم إضاعة وقت المستخدم وتركه ينتظر.

ساعدني إذا على حل هذه المشكلة...

تستطيع حل مشكلة البرنامج *Counter1.java* باستخدام النياسب. ضع المهمة الفرعية (أي الحلقة الموجودة في *go()*) داخل الطريقة *run()* الخاصة بالنيسب. وعندما يقوم المستخدم بضغط زر *Start* يعمل النيسب، ويتم بعدها إتمام إنشاء هذا النيسب. لذلك حتى لو كان النيسب فعالا، يمكن للبرنامج الرئيسي الاستمرار بالعمل. وإليك الحل:

```
//: Counter2.java
// A responsive user interface with threads
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
class SeparateSubTask extends Thread {
    private int count = 0;
    private Counter2 c2;
    private boolean runFlag = true;
    public SeparateSubTask(Counter2 c2) {
        this.c2 = c2;
        start();
    }
    public void invertFlag() { runFlag =
        !runFlag;}
    public void run() {
        while (true) {
            try {
```

```

        sleep(100);
    } catch (InterruptedException e) {}
    if(runFlag)
        c2.t.setText(Integer.toString(count++));
    }
}

public class Counter2 extends Applet {
    TextField t = new TextField(10);
    private SeparateSubTask sp = null;
    private Button
        onOff = new Button("Toggle"),
        start = new Button("Start");
    public void init() {
        add(t);
        start.addActionListener(new StartL());
        add(start);
        onOff.addActionListener(new OnOffL());
        add(onOff);
    }
    class StartL implements ActionListener {
        public void actionPerformed(ActionEvent
            e) {
            if(sp == null)
                sp = new SeparateSubTask(Counter2.this);
        }
    }
    class OnOffL implements ActionListener {
        public void actionPerformed(ActionEvent
            e) {
            if(sp != null)
                sp.invertFlag();
        }
    }
    public static void main(String[] args) {
        Counter2 applet = new Counter2();
        Frame aFrame = new Frame("Counter2");
        aFrame.addWindowListener(

```

```

new WindowAdapter() {
    public void windowClosing(WindowEvent
e) {
        System.exit(0);
    }
});
aFrame.add(applet, BorderLayout.CENTER);
aFrame.setSize(300,200);
applet.init();
applet.start();
aFrame.setVisible(true);
}
} ///:~

```

الآن وعندما يقوم المستخدم بضغط زر *start*، لا يتم استدعاء طريقة، وإنما يتم إنشاء نيسب للصف *SeparateSubTask*، وتستطيع حلقة الحدث *Counter2* الاستمرار بعد ذلك.

لاحظ أيضا بأنه يتم تخزين مؤشر العنصر *SeparateSubTask*، لذلك فإنه عندما يتم ضغط مفتاح *on/off* يتم فصل أو وصل *runFlag* داخل العنصر *SeparateSubTask*. وبعد ذلك يمكن للنيسب الاستمرار أو التوقف بنفسه. أما الصف *SeparateSubTask* فهو عبارة عن توسيع بسيط للصف *Thread* مع بان *constructor*، وطريقة *run()* لها نفس ترميز *go()* الموجود في البرنامج *Counter1.java*.

ويمكنك مشاركة المصادر المقيدة...

يمكن اعتبار برنامج بنيسب وحيد كأنه وحدة مستقلة تلف وتدور حول فضاء مشكلتك وتقوم بعمل وحيد في وقت معين. وبما أنه ليس هناك سوى وحدة مستقلة *entity* وحيدة فلن تحاول أبدا التفكير بمشكلة قيام وحدتين مستقلتين باستخدام نفس المصدر *resource* في نفس الوقت، مثل أن يقوم شخصان بمحاولة الجلوس في نفس المكان أو العبور من البوابة في نفس الوقت أو حتى التكلم في نفس الوقت.

لكن مع تعددية النياسب *multithreading* فسيكون بإمكانك استخدام أكثر من نيسب يقومون باستخدام نفس المصدر المقيد في نفس الوقت.

وتمتلك جافا العديد من الطرق من أجل منع التضارب على نمط واحد من المصادر كالذاكرة مثلا. وعلى اعتبار أنك تقوم بجعل عملية الوصول إلى عناصر معطيات صف من نمط خاص *private*، وتقوم بتحديد الوصول إلى الذاكرة من خلال الطرق فقط، تستطيع عندها منع التضارب بإنشاء طريقة خاصة *synchronized*. ويمكن لنيسب وحيد استدعاء الطريقة *synchronized* على عنصر محدد وفي وقت معين. ومن الأمثلة البسيطة عن هذه الطريقة:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

يحتوي كل عنصر على قفل *lock* وحيد يعتبر جزءا من هذا العنصر. وعندما تقوم باستدعاء أي طريقة *synchronized* يتم قفل هذا العنصر ولا يمكن استدعاء أية طريقة *synchronized* أخرى لهذا العنصر حتى تنتهي الطريقة الأولى وتحرر القفل.

فمثلا إذا افترضنا أننا استدعينا الطريقة *f()* في المثال السابق على عنصر، عندها لن نتمكن من استدعاء الطريقة *g()* على نفس العنصر حتى تنتهي الطريقة *f()* وتحرر القفل. لذلك فإن هناك قفلا وحيدا يمكن مشاركته بين جميع طرق *synchronized*

الخاصة بعنصر معين، ويمنع هذا القفل من الكتابة على الذاكرة المشتركة من قبل أكثر من طريقة واحدة في نفس الوقت.

يوجد أيضا قفل وحيد لكل صف (كجزء من العنصر *Class* لهذا الصف)، لذلك تستطيع طرق *synchronized static* قفل بعضها من خلال المعطيات الساكنة *static* في هذا الصف.

في البرنامج التالي سنتحكم بالعدادات، وسنفترض أن كل نيسب يمتلك عدادين تتم زيادتهما وإظهارهما داخل *run()*. لنفترض أيضا أن لدينا نيسبا آخر للصف *Watcher* الذي يقوم بمراقبة العدادات لمعرفة فيما إذا كانت متكافئة دائما. الهدف من هذا البرنامج استخدام الطريقة *synchronized* لمنع الوصول المتعدد لمصدر خاص (العداد في مثالنا هنا):

```
//: Sharing2.java
// Using the synchronized keyword to prevent
// multiple access to a particular resource.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
class TwoCounter2 extends Thread {
    private boolean started = false;
    private TextField
        t1 = new TextField(5),
        t2 = new TextField(5);
    private Label l =
        new Label("count1 == count2");
    private int count1 = 0, count2 = 0;
    public TwoCounter2(Container c) {
        Panel p = new Panel();
        p.add(t1);
        p.add(t2);
        p.add(l);
        c.add(p);
    }
    public void start() {
        if(!started) {
```

```

        started = true;
        super.start();
    }
}

public synchronized void run() {
    while (true) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
        try {
            sleep(500);
        } catch (InterruptedException e){}
    }
}

public synchronized void synchTest() {
    Sharing2.incrementAccess();
    if(count1 != count2)
        l.setText("Unsynched");
}

}

class Watcher2 extends Thread {
    private Sharing2 p;
    public Watcher2(Sharing2 p) {
        this.p = p;
        start();
    }
    public void run() {
        while(true) {
            for(int i = 0; i < p.s.length; i++)
                p.s[i].synchTest();
            try {
                sleep(500);
            } catch (InterruptedException e){}
        }
    }
}

public class Sharing2 extends Applet {
    TwoCounter2[] s;
    private static int accessCount = 0;

```

```

private static TextField aCount =
    new TextField("0", 10);
public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCou
nt));
}
private Button
    start = new Button("Start"),
    observer = new Button("Observe");
private boolean isApplet = true;
private int numCounters = 0;
private int numObservers = 0;
public void init() {
    if(isApplet) {
        numCounters =

        Integer.parseInt(getParameter("size"));
        numObservers =
            Integer.parseInt(
                getParameter("observers"));
    }
    s = new TwoCounter2[numCounters];
    for(int i = 0; i < s.length; i++)
        s[i] = new TwoCounter2(this);
    Panel p = new Panel();
    start.addActionListener(new StartL());
    p.add(start);
    observer.addActionListener(new
ObserverL());
    p.add(observer);
    p.add(new Label("Access Count"));
    p.add(aCount);
    add(p);
}
class StartL implements ActionListener {
    public void actionPerformed(ActionEvent
e) {

```

```

    for(int i = 0; i < s.length; i++)
        s[i].start();
    }
}

class ObserverL implements ActionListener {
    public void actionPerformed(ActionEvent e)
    {
        for(int i = 0; i < numObservers; i++)
            new Watcher2(Sharing2.this);
    }
}

public static void main(String[] args) {
    Sharing2 applet = new Sharing2();
    // This isn't an applet, so set the flag
    and
    // produce the parameter values from args:
    applet.isApplet = false;
    applet.numCounters =
        (args.length == 0 ? 5 :
         Integer.parseInt(args[0]));
    applet.numObservers =
        (args.length < 2 ? 5 :
         Integer.parseInt(args[1]));
    Frame aFrame = new Frame("Sharing2");
    aFrame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent
            e) {
                System.exit(0);
            }
        });
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(350, applet.numCounters
    *100);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
}

```



```
} ~: ///
```

لاحظ بأن الطريقتين `run()` و `synchTest()` هما من نمط `synchronized`. فإذا قمت بإجراء التزامن `synchronize` على إحدى الطريقتين فقط، فإن الطريقة الأخرى حرة بتجاهل قفل العنصر ويمكن استدعاؤها بلا مشاكل.

*نقطة هامة هنا وهي: يجب أن تحاول كل طريقة الوصول إلى مصدر مشترك خرج من نمط `synchronized` وإلا فلن تعمل بشكل صحيح.

المسألة الجديدة التي ظهرت أيضا هي أن العنصر `Watcher2` لا يستطيع أبدا إلقاء نظرة ولو سريعة عما يجري في الجوار، لأن الطريقة `run()` أصبحت بكاملها متزامنة `synchronized`، وباعتبار أن `run()` تعمل دوما من أجل كل عنصر، فإن القفل سيبقى محكما دوما ولن نتمكن أبدا من استدعاء الطريقة `synchTest()`. يمكن ملاحظة ذلك لأن `accessCount` لن يتغير أبدا.

مانريده من هذا المثال هو إيجاد طريقة لعزل جزء من الترميز داخل `run()`. يسمى هذا الجزء بالمقطع الحرج `critical section` وتستطيع استخدام كلمة المفتاح `synchronized` بطرق مختلفة لتحديد المقطع الحرج. وتستخدم جافا الكتل المتزامنة `synchronized block` لدعم المقاطع الحرجة، في هذه الحالة يتم استخدام `synchronized` لتحديد العنصر الذي استخدم قفله لتحديد تزامن الترميز التالي:

```
synchronized(syncObject) {
// This code can be accessed by only
// one thread at a time, assuming all
// threads respect syncObject's lock
}
```

وقبل أن يتم الدخول إلى الكتلة المتزامنة، يجب أن يتم قفل `syncObject`. فإذا امتلك أي نيسب آخر هذا القفل، فلن نتمكن من الدخول إلى الكتلة حتى نتخلى عن هذا القفل. ويمكن تعديل البرنامج السابق بإلغاء كلمة المفتاح `synchronized` من كامل الطريقة `run()` ووضع الكتلة `synchronized` حول السطرين الحرجين بدلا عن ذلك. لكن ماهو العنصر الواجب استخدامه كقفل؟ إنه القفل الموجود في `synchTest()` والذي هو العنصر الحالي (`this`) لذلك ستصبح الطريقة `run()` على الشكل:

```

public void run() {
    while (true) {
        synchronized(this) {
            t1.setText(Integer.toString(count1++));
            t2.setText(Integer.toString(count2++));
        }
        try {
            sleep(500);
        } catch (InterruptedException e){}
    }
}

```

بالطبع فإن جميع عمليات التزامن تعتمد على اجتهاد المبرمج، لكن يجب تغليف أي جزء من الترميز الذي يستطيع الوصول إلى مصدر مشترك ضمن الكتلة المتزامنة المناسبة.

ما هي حالات النيسب؟

يمكن للنيسب أن يكون في إحدى الحالات الأربع التالية:

١. إنشاء *New* : يمكن أن يتم إنشاء عنصر نيسب دون أن يتم تشغيله، لذلك لن يتمكن من العمل.
٢. قابل للعمل *Runnable* : أي أنه يمكن للنيسب العمل عندما تكون دورات المعالج *CPU cycles* متاحة له. لذلك يمكن للنيسب أن يكون عاملاً أو قد لا يكون، لكن ليس هناك أي شيء يمنع عمله إذا استطاع برنامج الجدولة *scheduler* تنظيم ذلك. في هذه الحالة فإن النيسب غير مجمد *blocked* أو ميت *dead*.
٣. ميت *Dead* : الحالة العادية التي تجعل نيسباً ما ميتاً هي بالإرجاع من طريقة *run()* الخاصة بهذا النيسب. بإمكانك أيضاً استدعاء الطريقة *stop()*، لكن ذلك سيؤدي إلى قذف استثناء عبارة عن صف فرعي من الصف *Error* (هذا يعني أنك لن تتمكن من التقاطه).
٤. مجمد *Blocked* : في هذه الحالة يمكن للنيسب العمل لكن هناك شيئاً ما يمنعه. وعندما يكون النيسب في هذا الوضع سيقوم برنامج الجدولة بالقفز عنه ببساطة ولن

يمنحه شيئاً من وقت المعالج. طبعاً لن يستطيع النيسب القيام بأي عمل قبل أن يعود إلى حالته العاملة *Runnable*.

لكن ماهي الأسباب التي تجعلنا نقوم بتجميد نيسب؟

هناك خمسة أسباب تتطلب تجميد النيسب وهي:

١. قمت بوضع النيسب في حالة نوم باستدعاء *sleep(milliseconds)* بحيث لن يتمكن من العمل في الوقت المحدد.
٢. قمت بتعليق تنفيذ نيسب باستخدام الطريقة *suspend()*، ولن يتمكن النيسب من العمل بعد ذلك قبل أن يتلقى رسالة الطريقة *resume()*.
٣. قمت بتعليق تنفيذ النيسب باستخدام الطريقة *wait()*، ولن يتمكن النيسب من العمل بعد ذلك قبل أن يتلقى رسالة الطريقة *notify()* أو *notifyAll()*.
٤. ينتظر النيسب إنهاء بعض عمليات الدخل والخرج.
٥. يقوم النيسب بمحاولة استدعاء الطريقة *synchronized* على عنصر آخر قبله غير متاح.

يمكنك أيضاً استدعاء الطريقة *yield()* (إحدى طرق الصف *Thread*) من أجل إعطاء زمن المعالج لنيسب أخرى طواعية لتتمكن من العمل. ويمكن أن يحدث نفس الشيء إذا وجد برنامج الجدولة أن نيسبك يمتلك الوقت الكافي وقرر القفز إلى نيسب آخر. لذلك فلن يكون هناك مانعاً هذا من إعادة برنامج الجدولة تشغيل نيسبك.

يوضح المثال التالي الطرق الخمس السابقة والتي تؤدي إلى تجميد نيسب. وسنقوم بفحص واختبار كل طريقة على حدة، مع أنها موجودة في ملف وحيد اسمه

Blocking.java، ولنبدأ أولاً بالبرنامج الرئيسي:

```
//: Blocking.java
// Demonstrates the various ways a thread
// can be blocked.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```

import java.io.*;
////////// The basic framework //////////
class Blockable extends Thread {
    private Peeker peeker;
    protected TextField state = new
        TextField(40);
    protected int i;
    public Blockable(Container c) {
        c.add(state);
        peeker = new Peeker(this, c);
    }
    public synchronized int read() { return i; }
    protected synchronized void update() {
        state.setText(getClass().getName()
            + " state: i = " + i);
    }
    public void stopPeeker() {
        // peeker.stop(); Deprecated in Java 1.2
        peeker.terminate(); // The preferred
            approach
    }
}

class Peeker extends Thread {
    private Blockable b;
    private int session;
    private TextField status = new
        TextField(40);
    private boolean stop = false;
    public Peeker(Blockable b, Container c) {
        c.add(status);
        this.b = b;
        start();
    }
    public void terminate() { stop = true; }
    public void run() {
        while (!stop) {
            status.setText(b.getClass().getName()
                + " Peeker " + (++session)

```

```

+ "; value = " + b.read());
try {
    sleep(100);
} catch (InterruptedException e) {}
}
}
} ///:Continued

```

من الجزء السابق سنعتبر الصف *Blockable* صفا أساسيا لجميع الصفوف. وكل عنصر في هذا الصف سيحتوي على *TextField* اسمه *state* سيتم استخدامه لإظهار معلومات عن العنصر. أما الطريقة التي سنقوم بإظهار هذه المعلومات فهي *update()*. وهي كما ترى تستخدم *getClass().getNmae()* من أجل توليد اسم الصف بدلا من طباعته فقط، لأن الطريقة *update()* لا تستطيع معرفة الاسم الصحيح للصف الذي قام بطلبها، لأنه صف مشتق من الصف *Blockable*. أما العنصر الذي سيدل على إجراء التغييرات في الصف *Blockable* فهو *int I* الذي تتم زيادة قيمته من خلال الطريقة *run()* في الصف المشتق. وهناك نيسب للصف *Peeker* يبدأ من أجل كل عنصر *Blockable*، أما عمل عنصر الصف *Peeker* فهو مراقبة عنصر *Blockable* الموافق لرؤية التغييرات على *i* وذلك باستدعاء الطريقة *read()* ووضعها في *TextField status* الخاص بها.

```

الاختبار الأول في هذا البرنامج سيكون مع الطريقة sleep():
///:Continuing
////////// Blocking via sleep() //////////
class Sleeper1 extends Blockable {
    public Sleeper1(Container c) { super(c); }
    public synchronized void run() {
        while(true) {
            i++;
            update();
            try {
                sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}

```

```

    }
}
class Sleeper2 extends Blockable {
    public Sleeper2(Container c) { super(c); }
    public void run() {
        while(true) {
            change();
            try {
                sleep(1000);
            } catch (InterruptedException e){}
        }
    }
    public synchronized void change() {
        i++;
        update();
    }
}
} ///:Continued

```

في الصف *Sleeper1* فإن الطريقة *run()* بكاملها متزامنة *synchronized*. وستجد بأن عنصر *Peeker* المرتبط بهذا العنصر سيستمر بالعمل حتى تقوم بتشغيل النيسب عندها سيتوقف هذا العنصر عن العمل. وهذا هو أحد أشكال التجميد *blocking* لأن *Sleeper1.run()* متزامن *synchronized*، وعندما يبدأ النيسب بالعمل فسيبقى دائما داخل *run()*، ولن تتمكن هذه الطريقة من الحصول على قفل العنصر وبالتالي سيجمد عنصر *Peeker*.

وهنا يعطينا الصف *Peeker2* الحل وذلك بالعمل بالنمط غير المتزامن *unsynchronized*، و فقط الطريقة *change()* تبقى متزامنة *synchronized*. هذا يعني بأنه كلما كانت الطريقة *run()* في حالة *sleep()* سيتمكن عنصر *Peeker* من الوصول إلى طريقة *synchronized* التي يحتاجها والمسماة *read()*. ستري هنا بأن عنصر *Peeker* سيتابع عمله عندما تبدأ بتشغيل نيسب *Sleeper2*.

الجزء التالي من هذا المثال يوضح مفهوم التعليق *suspension*. والصف *Thread* يمتلك الطريقة *suspend()* من أجل إيقاف النيسب مؤقتا وطلب الطريقة

`resume()` التي تقوم بإعادة تشغيله من النقطة التي توقف عندها. وبشكل افتراضي

يتم استدعاء الطريقة `resume()` من خلال نياش أخرى غير النياش المعلق:

```

///:Continuing
////////// Blocking via suspend() //////////
class SuspendResume extends Blockable {
    public SuspendResume(Container c) {
        super(c);
        new Resumer(this);
    }
}

class SuspendResume1 extends SuspendResume {
    public SuspendResume1(Container c) {
        super(c);
    }
    public synchronized void run() {
        while(true) {
            i++;
            update();
            suspend(); // Deprecated in Java 1.2
        }
    }
}

class SuspendResume2 extends SuspendResume {
    public SuspendResume2(Container c) {
        super(c);
    }
    public void run() {
        while(true) {
            change();
            suspend(); // Deprecated in Java 1.2
        }
    }
    public synchronized void change() {
        i++;
        update();
    }
}

class Resumer extends Thread {
    private SuspendResume sr;

```

```

public Resumer(SuspendResume sr) {
    this.sr = sr;
    start();
}
public void run() {
    while(true) {
        try {
            sleep(1000);
        } catch (InterruptedException e){}
        sr.resume(); // Deprecated in Java 1.2
    }
}
} ///:Continued

```

هنا أيضا يمتلك الصف *SuspendResume1* الطريقة *synchronized* *run()* وعندما تقوم بتشغيل النيسب سترى بأن عنصر *Peeker* الموافق سيجد بانتظار أن يصبح القفل متاحا، وهذا لن يحدث أبدا.

وتم حل هذه المشكلة في الصف *SuspendResume2* الذي لايقوم بإجراء التزامن *synchronize* على كامل الطريقة *run()*، وإنما يقوم باستخدام الطريقة *synchronize change()* بدلا عنها.

وكما ترى فإن النقطة الأساسية في الجزأين السابقين، هي أن كلا الطريقتين *sleep()* و *suspend()* لايقومان بتحرير القفل عند استدعائهما، كذلك فإن الطريقة *wait()* لايقوم بتحرير القفل عند استدعائها، مما يعني أنه يمكن استدعاء الطرق المتزامنة الأخرى *synchronized* في عنصر النيسب وذلك من خلال الطريقة *wait()* سترى في الصفيين الجديدين بأن الطريقة *run()* ستكون متزامنة *synchronized* فيهما، وسيبقى الصف *Peeker* قادرا على الوصول الكامل إلى الطرق المتزامنة من خلال الطريقة *wait()* التي تقوم بتحرير القفل من العنصر لأنها تقوم بتعليق الطريقة التي قامت باستدعائها.

سترى أيضا بأن هناك شكلين للطريقة *wait()* : الشكل الأول يأخذ وسيطا بالميلي ثانية *milliseconds* كما في الطريقة *sleep()*، والهدف منه التوقف لفترة من الزمن.

أما الشكل الثاني فلا يأخذ أي وسطاء، مما يعني بأن الطريقة `wait()` تستطيع متابعة العمل حتى تأتي الطريقة `notify()` وتبدأ العمل.

الأمر الهام الذي يجب الانتباه إليه هنا هو أن الطريقتين `wait()` و `notify()` جزءان من الصف `Object` (وليس من الصف `Thread` مثل الطرق `sleep()` و `suspend()` و `resume()`). وقد يبدو الأمر غريبا بعض الشيء إلا أن السبب في ذلك هو أن القفل هو جزء من أي عنصر. و بالنتيجة يمكنك وضع الطريقة `wait()` داخل أي طريقة متزامنة `synchronized` بغض النظر عن وجود أي نيسب داخل هذا الصف الخاص.

وفي الحقيقة فالمكان الوحيد الذي تستطيع استدعاء الطريقة `wait()` من خلاله هو داخل طريقة متزامنة `synchronized method` أو كتلة `block`. وإذا قمت باستدعاء الطريقة `wait()` أو `notify()` ضمن طريقة غير متزامنة فيمكن ترجمة البرنامج إلا أنك عند ما تقوم بتنفيذ البرنامج ستحصل على الاستثناء `IllegalMonitorStateException`.

وبإمكانك استدعاء الطريقة `wait()` أو `notify()` من أجل القفل المتعلق بك فقط. باستطاعتك هنا أيضا ترجمة البرنامج لكنك ستحصل على نفس الاستثناء السابق. من أجل ذلك تستطيع إنشاء طريقة متزامنة `synchronized` تقوم بدورها باستدعاء الطريقة `notify()` إلى العنصر الخاص بها فقط. ومع ذلك يمكنك استدعاء الطريقة `notify()` في الصف `Notifier` وذلك ضمن كتلة متزامنة:

```
synchronized(wn2) {
    wn2.notify();
}
```

حيث أن `wn2` هو عنصر من نمط `WaitNotify2`. ويمكن لهذه الطريقة الحصول على قفل للعنصر `wn2`، مما يسمح لك باستدعاء الطريقة `notify()` على العنصر `wn2` و بلا مشاكل:

```
///  
Continuing  
////////// Blocking via wait() //////////  
class WaitNotify1 extends Blockable {  
    public WaitNotify1(Container c) { super(c); }
```

```

public synchronized void run() {
    while(true) {
        i++;
        update();
        try {
            wait(1000);
        } catch (InterruptedException e){}
    }
}

class WaitNotify2 extends Blockable {
    public WaitNotify2(Container c) {
        super(c);
        new Notifier(this);
    }
    public synchronized void run() {
        while(true) {
            i++;
            update();
            try {
                wait();
            } catch (InterruptedException e){}
        }
    }
}

class Notifier extends Thread {
    private WaitNotify2 wn2;
    public Notifier(WaitNotify2 wn2) {
        this.wn2 = wn2;
        start();
    }
    public void run() {
        while(true) {
            try {
                sleep(2000);
            } catch (InterruptedException e){}
            synchronized(wn2) {
                wn2.notify();
            }
        }
    }
}

```

```

    }
    }
}
} ///:Continued

```

تسمح لك الطريقة `wait()` بوضع النيسب في حالة نوم `sleep` بانتظار أن يتغير العالم، ولا تجعله يستيقظ إلا عند ظهور الطريقة `notify()` أو الطريقة `notifyAll()` حيث يقوم بتفحص واختبار التغييرات. وهذا بالطبع يساعد على إجراء التزامن بين النياسب.

لننتقل الآن إلى مشكلة تجميد الدخل والخرج. في الجزء التالي من مثالنا ستجد بأن الصنفين السابقين يعملان مع عناصر `Reader` و `Writer`. أما الصنف `Sender` فيقوم بوضع المعطيات ضمن `Writer` وينام لفترة عشوائية من الوقت. وعلى الرغم من أن الصنف `Receiver` لا يمتلك الطرق `sleep()` و `suspend()` و `wait()` إلا أنه يجمد تلقائياً عند استدعاء الطريقة `read()` وعندما لا تبقى هناك أية معطيات.

```

///:Continuing
class Sender extends Blockable { // send
    private Writer out;
    public Sender(Container c, Writer out) {
        super(c);
        this.out = out;
    }
    public void run() {
        while(true) {
            for(char c = 'A'; c <= 'z'; c++) {
                try {
                    i++;
                    out.write(c);
                    state.setText("Sender sent: "
                        + (char)c);
                    sleep((int) (3000 * Math.random()));
                } catch (InterruptedException e) {}
                catch (IOException e) {}
            }
        }
    }
}

```

```

}
class Receiver extends Blockable {
    private Reader in;
    public Receiver(Container c, Reader in) {
        super(c);
        this.in = in;
    }
    public void run() {
        try {
            while(true) {
                i++; // Show peeker it's alive
                // Blocks until characters are there:
                state.setText("Receiver read: "
                    + (char)in.read());
            }
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
} ///:Continued

```

كلا الصغين يقومان بوضع المعلومات ضمن حقليهما *state* وبتغيير *i* بحيث يتمكن عنصر *Peeker* من رؤية النيسب وهو يقوم بعمله.

أخيرا فإن صف البريمج الأساسي بسيط جدا، لأنه تم إنجاز أغلب العمل ضمن جزء *Blockable*. سنقوم بإنشاء مصفوفة من عناصر *Blockable*، وعلى اعتبار أن كل عنصر في هذه المصفوفة عبارة عن نيسب، فإنها تتمكن من إنجاز أعمالها بنفسها عندما تقوم بالضغط على زر *start*. هناك أيضا زر وعبارة *actionPerformed()* من أجل إيقاف جميع عناصر *Peeker*.

ومن أجل تثبيت الاتصال بين عناصر *Sender* و *Receiver* يتم إنشاء *PipedWriter* و *PipedReader*. لاحظ هنا بأنه يجب وصل *PipedReader in* مع *PipedWriter out* من خلال وسيط البناء.

```

///:Continuing
////////// Testing Everything //////////
public class Blocking extends Applet {
    private Button

```

```

    start = new Button("Start") ,
    stopPeekers = new Button("Stop Peekers");
    private boolean started = false;
    private Blockable[] b;
    private PipedWriter out;
    private PipedReader in;
    public void init() {
        out = new PipedWriter();
        try {
            in = new PipedReader(out);
        } catch(IOException e) {}
        b = new Blockable[] {
            new Sleeper1(this),
            new Sleeper2(this),
            new SuspendResume1(this),
            new SuspendResume2(this),
            new WaitNotify1(this),
            new WaitNotify2(this),
            new Sender(this, out),
            new Receiver(this, in)
        };
        start.addActionListener(new StartL());
        add(start);
        stopPeekers.addActionListener(
            new StopPeekersL());
        add(stopPeekers);
    }
    class StartL implements ActionListener {
        public void actionPerformed(ActionEvent
            e) {
            if(!started) {
                started = true;
                for(int i = 0; i < b.length; i++)
                    b[i].start();
            }
        }
    }
}

```

```

class StopPeekersL implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        // Demonstration of the preferred
        // alternative to Thread.stop():
        for(int i = 0; i < b.length; i++)
            b[i].stopPeeker();
    }
}

public static void main(String[] args) {
    Blocking applet = new Blocking();
    Frame aFrame = new Frame("Blocking");
    aFrame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent
                e) {
                System.exit(0);
            }
        });
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(350,550);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
}
} ///:~

```

لاحظ كيف تقوم الحلقة بعبور كامل المصفوفة ضمن الطريقة `init()` وإضافة الحالة `state` و الحقول النصية `peeker.status` إلى الصفحة.

لاحظ أيضا أنه بعد إنشاء نياسب الصف `Blockable` كيف يقوم كل منها بإنشاء وتشغيل عنصر `Peeker` الخاص به وبشكل تلقائي. لذلك سترى عنصر `Peekers` يعمل قبل بدء تشغيل نياسب `Blockable`. وهو أمر هام لأن بعض عناصر `Peekers` ستجمد وتتوقف عند بدء عمل نياسب `Blockable`.

هناك أفضليات للنiasب...

تخبر أفضليات *priority* النiasب برنامج الجدولة *scheduler* عن أهمية كل نiasب. فإذا كان هناك عدد من النiasب المجمدة والتي تنتظر بدء العمل، يقوم برنامج الجدولة بتشغيل النiasب ذو الأفضلية الأعلى أولا. وبالطبع هذا لايعني بأن النiasب ذو الأفضلية الأدنى لن يعمل أبدا لكنه سيعمل بشكل أقل. يمكنك قراءة أفضلية النiasب باستخدام الطريقة *getPriority()*، ومن أجل تغييرها يمكنك استخدام الطريقة *setPriority()*.

مجموعات النiasب...

يمكن أن تنتمي عدة نiasب إلى مجموعة نiasب *thread group*. ويمكن أن تكون هذه المجموعة إما المجموعة الافتراضية أو مجموعة يمكنك التصريح عنها عندما تقوم بإنشاء نiasب. وعند إنشاء نiasب ما يكون مقيدا بمجموعة ولايمكن أبدا تغييره إلى مجموعة أخرى. كما أن أي تطبيق يمتلك نiasبا واحدا على الأقل ينتمي إلى مجموعة نiasب النظام، وعندما تقوم بإنشاء نiasب أخرى دون تحديد المجموعة التي ستتنتمي إليها فإنها ستتوضع في مجموعة نiasب النظام.

كذلك يمكن أن تنتمي مجموعة نiasب إلى مجموعة نiasب أخرى. ومن الواجب تحديد مجموعة النiasب التي تنتمي إليها المجموعة الجديدة وذلك ضمن الباني *constructor*. وتنتمي جميع مجموعات النiasب إلى مجموعة نiasب النظام التي تعتبر المجموعة الأم.

والسبب الرئيسي لإنشاء مجموعات النiasب هو سبب أمني *security*، حيث يمكن للنiasب الموجودة ضمن مجموعة نiasب القيام بتعديل النiasب الأخرى في نفس المجموعة والنiasب الموجودة في المجموعات الأبناء، بينما لايمكنها التعديل على النiasب خارج

المجموعة. يوضح المثال التالي أن نيسبا ما في المجموعة الفرعية الورقة leaf subgroup يقوم بتعديل أفضليات جميع النيساب في شجرته.

```
//: TestAccess.java
// How threads can access other threads
// in a parent thread group
public class TestAccess {
    public static void main(String[] args) {
        ThreadGroup
            x = new ThreadGroup("x"),
            y = new ThreadGroup(x, "y"),
            z = new ThreadGroup(y, "z");
        Thread
            one = new TestThread1(x, "one"),
            two = new TestThread2(z, "two");
    }
}
class TestThread1 extends Thread {
    private int i;
    TestThread1(ThreadGroup g, String name) {
        super(g, name);
    }
    void f() {
        i++; // modify this thread
        System.out.println(getName() + " f()");
    }
}
class TestThread2 extends TestThread1 {
    TestThread2(ThreadGroup g, String name) {
        super(g, name);
        start();
    }
    public void run() {
        ThreadGroup g =
            getThreadGroup().getParent().getParent();
        g.list();
        Thread[] gAll = new
            Thread[g.activeCount()];
```



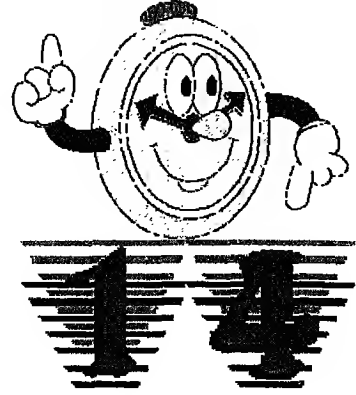
```

g.enumerate(gAll);
for(int i = 0; i < gAll.length; i++) {
    gAll[i].setPriority(Thread.MIN_PRIORITY);
    ((TestThread1)gAll[i]).f();
}
g.list();
}
} ///:~

```

لاحظ أنه ضمن الطريقة `main()`، يتم إنشاء عدة مجموعات ينسب لـ `ThreadGroups` : `x` لا يمتلك أي وسيط لذلك يتم وضعه تلقائياً ضمن مجموعة ينسب النظام، أما `y` فهو تابع لـ `x` و `z` تابع لـ `y`.

يوجد كذلك نيسبان يتم إنشاؤها ووضعها في مجموعتين مختلفتين. النيسب الأول `TestThread1` لا يمتلك الطريقة `run()` لكنه يحتوي على الطريقة `f()` التي تقوم بتعديل النيسب وطباعة بعض المعلومات. أما النيسب الثاني فهو `TestThread2` وهو صف فرعي من النيسب `TestThread1`. وتقوم الطريقة `run()` في النيسب الفرعي بأخذ مجموعة النيسب الحالي، ثم تعديل توريث الشجرة بمستويين باستخدام الطريقة `getParent()`. يتم بعدها إنشاء مصفوفة مؤشرات إلى `Thread` باستخدام الطريقة `activeCount()` وذلك من أجل السؤال عن عدد النيسب في مجموعة النيسب هذه والمجموعات الأبناء. أما الطريقة `enumerate()` فتقوم بوضع مؤشرات لجميع هذه النيسب في المصفوفة `gAll`. بعدها يقوم العضو `i` بالانتقال ضمن كامل المصفوفة واستدعاء الطريقة `f()` لكل نيسب إضافة إلى تعديل الأفضلية.



كما تعلم فلقد كانت برمجة الشبكات عملية صعبة ومعقدة، وكان يتوجب على المبرمج معرفة الكثير من التفاصيل عن الشبكات وحتى عن التجهيزات المادية في أغلب الأحيان.

بالطبع يتوجب عليك فهم مختلف طبقات بروتوكول الشبكة، ومعرفة الكثير عن الدالات الخاصة بكل مكتبة شبكة والمتعلقة بالاتصال، وتحريم *packing* وفك تحريم *unpacking* كتل المعلومات.

تعتبر برمجة الشبكات إحدى الميزات القوية التي تميز لغة جافا، ولقد حاولت قدر الإمكان تجريد التفاصيل المختلفة والخاصة بالشبكة وعالجتها ضمن آلة جافا الوهمية *JVM*. ويجري تغليف اتصال الشبكة بعناصر دفق *stream objects*، لذلك فأنت تقوم في النهاية باستدعاء نفس الطرق مع بقية أنواع الدفق. إضافة إلى ذلك فإن تعددية النيايب *multithreading* المبنية ضمن جافا تساعد كثيراً عند معالجة شبكات أخرى، هذا يفيد في معالجة العديد من الاتصالات في نفس الوقت. سنقوم في هذا الفصل بإيضاح كيفية دعم جافا للشبكات عن طريق العديد من الأمثلة السهلة.

عليك أولاً تعريف جهازك...

من أجل التفريق بين جهاز وآخر والتأكد من أنك قمت بالاتصال مع الجهاز الذي تريد، فعليك تعريف كل جهاز ضمن الشبكة وإعطائه محدداً وحيداً *unique identifier*. وعلى اعتبار أن طريقة إعطاء اسم وحيد ضمن شبكة محلية أصبحت غير كافية خاصة وأن جافا تتعامل مع شبكة الإنترنت، فلقد أصبح من الضروري إيجاد طريقة تقوم بإعطاء محدّد وحيد لكل جهاز يختلف عن أيّ محدّد آخر في العالم. لذلك جرى استخدام طريقة عنوان بروتوكول إنترنت *IP (Internet Protocol)* التي تأخذ أحد الشكلين التاليين:

١. شكل *DNS (Domain Name Service)* الشائع. فمثلاً إذا كان اسم المجال *noukari.com*، وإذا افترضنا أن لدينا الجهاز *Mirna* ضمن هذا المجال، فإن اسم المجال سيصبح على الشكل: *Mirna.noukari.com* وهو نفس طريقة التسمية المستخدمة في البريد الإلكتروني *Email*.

٢. أما الشكل الثاني فهو عبارة عن أرقام تمثل العناوين ونفصل بينها بنقاط مثلاً:

132.255.28.120

في كلتا الحالتين يمكن تمثيل عنوان إنترنت IP برقم 32-bit، مما يساعد على توليد عدد خرافي من الأرقام. ويمكن استخدام عنصر جافا خاصاً من أجل تمثيل الرقم الناتج عن أحد الشكليات السابقين باستخدام الطريقة `static InetAddress.getByName()` الموجودة في المكتبة `java.net`، أما النتيجة فهي عبارة عن عنصر من نمط `InetAddress` يمكنك استخدامه لبناء مقبس `socket` كما سنرى لاحقاً.

يوضح المثال التالي كيفية استخدام الطريقة السابقة من أجل توليد عنوان إنترنت IP. لاستخدام هذا المثال عليك معرفة اسم جهازك (تم اختباره ضمن نظام Windows95 فقط):

```
//: WhoAmI.java
// Finds out your network address when you're
// connected to the Internet.
package c15;
import java.net.*;
public class WhoAmI {
    public static void main(String[] args)
        throws Exception {
        if(args.length != 1) {
            System.err.println(
                "Usage: WhoAmI MachineName");
            System.exit(1);
        }
        InetAddress a =
            InetAddress.getByName(args[0]);
        System.out.println(a);
    }
} ///:~
```

المقابس Sockets ...

المقبس *socket* عبارة عن برنامج يستخدم لتمثيل أطراف *Terminals* الاتصال بين جهازين. وفي حال وجود اتصال، فهناك مقبس على كل جهاز ويمكنك تخيل وجود كبل افتراضي يصل بين الجهازين عند كل مقبس.

بالطبع فإنّ العتاد الفيزيائي و الكابلات بين الأجهزة ستكون غير مهمة لنا ولا يتوجب علينا إلا معرفة الضروري عنها.

في لغة جافا، وعندما تقوم بإنشاء مقبس *socket* لإنشاء اتصال مع أجهزة أخرى، ستحصل على عناصر من نمط *Reader* أو *Writer* من أجل معالجة الاتصال كعنصر دفق دخل أو خرج. ويوجد صنفان يمثلان المقابس: الأول *ServerSocket* يستخدمه المخدم *Server* لسماع الاتصالات الواردة، أما الثاني فهو الصنف *Socket* ويستخدمه الزبون *Client* لتمهيد الاتصال. وحالما يقوم الزبون بإنشاء اتصال مقبس، يرجع عنصر *ServerSocket* (من خلال الطريقة *accept()*) إلى جهة المخدم الموافق لعنصر *Socket* حتى يتم البدء بالاتصال المباشر. ستحصل بعد ذلك على اتصال حقيقي من مقبس *Socket* إلى مقبس *Socket* ويمكنك معالجة كلتا النهايتين بنفس الطريقة.

ابتداءً من هذه النقطة، يمكنك استخدام الطريقتين *GetReader()* أو *GetWriter()* لتوليد عناصر *Reader* أو *Writer* في كل مقبس *Socket*. ويقوم *ServerSocket* بإنشاء مخدم فيزيائي *physical server* أو مقبس مستمع *listening socket* على الجهاز المضيف *host machine*. ويقوم هذا المقبس بالتصّنت على الاتصالات الواردة وإرجاع مقبس مثبت *established socket* من خلال الطريقة *accept()*. ومن الغريب هنا هو أنّ كلا المقبسين (المستمع والمثبت) يمكنهما الاتصال مع نفس مقبس المخدم *server socket*. ويمكن للمقبس المستمع قبول طلبات الاتصال الجديدة فقط ولا يمكنه قبول حزم المعطيات *Data Packet*. لذلك وعلى الرغم من أنّ *ServerSocket* لا يقوم برمجياً بأشياء كثيرة إلا أنّه فيزيائياً ينجز أموراً هامة.

وعندما نقوم بإنشاء `ServerSocket` عليك إعطائه رقم بوابة `port number` فقط، ولست بحاجة إلى إعطائه عنوان إنترنت `IP Address` لأنه موجود أصلاً في الجهاز الذي يمثلّه. لكن عندما نقوم بإنشاء عنصر `Socket` عليك إعطائه عنوان إنترنت `IP` ورقم البوابة التي تحاول الاتصال بها.

التعامل مع مخدّم/زبون بسيط...

سنقوم في المثال القادم بتوضيح الاستخدام الأبسط للمخدّم/الزبون باستخدام المقابس. كل مايفعله المخدّم هنا هو انتظار اتصال، وعند حصول هذا الاتصال يستخدم المقبس `Socket` الناتج عن الاتصال لإنشاء عنصري الدخل والخرج `Reader` و `Writer`. بعد ذلك فإن أي شيء يتم قراءته من `Reader` يتم إرساله إلى `Writer` حتى يصل إلى السطر `END` حيث تكون نهاية الاتصال.

ويقوم الزبون بإنشاء الاتصال مع المخدّم ثم إنشاء عنصر `Writer`، الذي يقوم بإرسال أسطر نصيّة. يقوم الزبون أيضاً بإنشاء عنصر `Reader` لسماع مايقوله المخدّم. ويستخدم المخدّم والزبون نفس رقم البوابة، كما يستخدم الزبون عنوان الانقلاب العودي المحلي `local loopback address` للاتصال بالمخدّم على نفس الجهاز، لذلك لن تكون بحاجة إلى اختباره على الشبكة.

فلنبدأ أولاً بالمخدّم `Server`:

```
//: JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;
public class JabberServer {
    // Choose a port outside of the range 1-1024:
    public static final int PORT = 8080;
    public static void main(String[] args)
        throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        System.out.println("Started: " + s);
```

```

try {
    // Blocks until a connection occurs:
    Socket socket = s.accept();
    try {
        System.out.println(
            "Connection accepted: "+ socket);
        BufferedReader in =
            new BufferedReader(
                new InputStreamReader(
                    socket.getInputStream()));
        // Output is automatically flushed
        // by PrintWriter:
        PrintWriter out =
            new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(
                        socket.getOutputStream()))), true);
        while (true) {
            String str = in.readLine();
            if (str.equals("END")) break;
            System.out.println("Echoing: " + str);
            out.println(str);
        }
        // Always close the two sockets...
    } finally {
        System.out.println("closing...");
        socket.close();
    }
} finally {
    s.close();
}
}
} ///:~

```

كما تلاحظ فإن `ServerSocket` يحتاج فقط إلى رقم بوابة، ولا يحتاج إلى عنوان إنترنت `IP`. وعندما نقوم باستدعاء الطريقة `accept()`، تجدد الطريقة حتى يحاول بعض الزبائن الاتصال بها. وعند حدوث اتصال، تقوم الطريقة `accept()` بإرجاع



عنصر *Socket* يمثل هذا الاتصال. وتجرى طباعة كلا العنصرين *ServerSocket* و *Socket* الناتجان عن الطريقة *accept()* في *System.out*. مما يعني أنه يجري استدعاء طرق *toString()* الخاصة بهما بشكل تلقائي. وهذا يولد:

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

سترى لاحقاً كيف سيتم ربط ذلك مع مايفعله الزبون.

يقوم الجزء التالي من البرنامج بفتح الملفات للقراءة والكتابة وتوليد *InputStream* و *OutputStream* من العنصر *Socket*. ويتم تحويل كلا العنصرين *InputStream* و *OutputStream* إلى عناصر *Java 1.2 Reader* و *Writer* باستخدام صفوف القالب *InputStreamReader* و *OutputStreamReader*.

ونقوم الحلقة اللانهائية بقراءة أسطر من *BufferedReader in* وكتابة معلومات ضمن *System.out* و *PrintWriter out*. وعندما يقوم الزبون بإرسال سطر يحتوي على كلمة *END* يخرج البرنامج من الحلقة ويغلق المقبس *Socket*. لنستعرض معاً برنامج الزبون *Client*:

```
//: JabberClient.java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;
public class JabberClient {
    public static void main(String[] args)
        throws IOException {
        // Passing null to getByName() produces the
        // special "Local Loopback" IP address, for
        // testing on one machine w/o a network:
        InetAddress addr =
            InetAddress.getByName(null);
        // Alternatively, you can use
```

```
// the address or name:
// InetAddress addr =
// InetAddress.getBy_name("127.0.0.1");
// InetAddress addr =
// InetAddress.getBy_name("localhost");
System.out.println("addr = " + addr);
Socket socket =
    new Socket(addr, JabberServer.PORT);
// Guard everything in a try-finally to
make
// sure that the socket is closed:
try {
    System.out.println("socket = " + socket);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(
                socket.getInputStream()));
    // Output is automatically flushed
    // by PrintWriter:
    PrintWriter out =
        new PrintWriter(
            new BufferedWriter(
                new OutputStreamWriter(
                    socket.getOutputStream()))), true);
    for(int i = 0; i < 10; i++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
    }
    out.println("END");
} finally {
    System.out.println("closing...");
    socket.close();
}
}
} ///:~
```

في البرنامج السابق، وضمن جزء `main()`، يمكنك رؤية كيفية استخدام الطرق الثلاث جميعها والتي تساعد على توليد عنوان إنترنت المحلي `local loopback IP address` وذلك باستخدام `null` أو `localhost` أو العنوان الصريح المحجوز `127.0.0.1` (عنوان `IP` المحلي). بالطبع عندما ترغب بالاتصال مع جهاز عبر الشبكة يجب عليك استبداله بعنوان إنترنت الخاص بهذا الجهاز. وعندما تتم طباعة `InetAddress Addr` ستظهر النتيجة:

```
localhost/127.0.0.1
```

لاحظ أنه يتم إنشاء المقبس المسمى `socket` باستخدام `InetAddress` ورقم البوابة. ولفهم معنى ذلك، تذكر بأن اتصال إنترنت محدّد بقطع المعطيات الأربع هذه: `ClientHost`, `clientPortNumber`, `serverHost`, `serverPortNumber`. ويستحوذ المخدم على البوابة `8080` ضمن المضيف المحلي `localhost (127.0.0.1)`. أما الزبون فيقوم بحجز البوابة التالية المتاحة على جهازه (`1077` في حالتنا هنا)، وهو ما يحدث أيضاً على الجهاز `(127.0.0.1)`.

حتى يجري نقل المعطيات بين المخدم والزبون، يتوجّب على كل طرف معرفة مكان إرسالها. لذلك يقوم الزبون بإرجاع عنوانه عند عملية الاتصال بالمخدم، حتى يستطيع

المخدم تحديد مكان إرسال المعطيات. انظر إلى خرج البرنامج السابق في موقع المخدم:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

هذا يعني بأن المخدم يقوم فقط بقبول الاتصال من `127.0.0.1` على البوابة `1077`

عند استماعه على بوابته المحلية `8080`. أما في موقع الزبون فسترى الخرج التالي:

```
Socket[addr=localhost/127.0.0.1,PORT=8080,local  
port=1077]
```

مما يعني بأن الزبون يقوم بإقامة اتصال مع `127.0.0.1` على البوابة `8080` باستخدام البوابة المحلية `1077`.

تقديم عدة زبائن في نفس الوقت!!!؟

توضح البرامج السابقة أنه بإمكانك تقديم زبون واحد فقط في وقت معين. لكن في الحالة العادية، يتوجب على المخدم تقديم عدة زبائن في نفس الوقت. وهنا يأتي دور تعددية النياسب *multithreading* التي قمنا بشرح أهميتها استخدامها في لغة جافا. الطريقة الأساسية للقيام بذلك تتمثل بإنشاء مقبس مخدم *ServerSocket* وحيد واستدعاء الطريقة *accept()* من أجل انتظار اتصال جديد. وعندما يتم الإرجاع من هذه الطريقة، يأخذ المقبس *Socket* الناتج ويستخدمه لإنشاء نيسب *thread* جديد من أجل تقديم زبون خاص، ثم يتم استدعاء *accept()* لانتظار زبون جديد. و ستلاحظ أن المثال التالي والخاص بالمخدم، يشبه المثال *JabberServer.java* كثيراً، إلا أنه تم تعديل جميع العمليات الخاصة بتقديم زبون خاص بحيث تم نقلها إلى صف نيسب منفصل:

```
//: MultiJabberServer.java
// A server that uses multithreading to handle
// any number of clients.
import java.io.*;
import java.net.*;
class ServeOneJabber extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    public ServeOneJabber(Socket s)
        throws IOException {
        socket = s;
        in =
            new BufferedReader(
                new InputStreamReader(
                    socket.getInputStream()));
        // Enable auto-flush:
        out =
            new PrintWriter(
                new BufferedWriter(
```



```

        new OutputStreamWriter(
            socket.getOutputStream()), true);
    // If any of the above calls throw an
    // exception, the caller is responsible for
    // closing the socket. Otherwise the thread
    // will close it.
    start(); // Calls run()
}
public void run() {
    try {
        while (true) {
            String str = in.readLine();
            if (str.equals("END")) break;
            System.out.println("Echoing: " + str);
            out.println(str);
        }
        System.out.println("closing...");
    } catch (IOException e) {
    } finally {
        try {
            socket.close();
        } catch (IOException e) {}
    }
}
}

public class MultiJabberServer {
    static final int PORT = 8080;
    public static void main(String[] args)
        throws IOException {
        ServerSocket s = new
        ServerSocket(PORT);
        System.out.println("Server Started");
        try {
            while(true) {
                // Blocks until a connection occurs:
                Socket socket = s.accept();
                try {
                    new ServeOneJabber(socket);

```

```

    } catch(IOException e) {
        // If it fails, close the socket,
        // otherwise the thread will close
        it:
        socket.close();
    }
}
} finally {
    s.close();
}
}
} ///:~

```

يقوم النيسب *ServeOneJabber* بأخذ المقبس الناتج عن *accept()* ضمن *main()*، وذلك في كل مرة يقوم فيها زبون جديد بإجراء اتصال. يقوم بعدها بإنشاء *BufferedReader* وتفرغ عنصر *PrintWriter* بشكل تلقائي باستخدام عنصر *Socket*. وأخيراً يقوم باستدعاء طريقة *start()* الخاصة بالصف *Thread* والتي تقوم بعمليات التمهيد للنيسب ومن ثم استدعاء الطريقة *run()*. يتم هنا إنجاز نفس نمط الأعمال الموجودة في المثال السابق: قراءة بعض الأشياء من المقبس ثم إظهارها حتى الحصول على إشارة *END*.

استخدام البروتوكول UDP...

قمنا في الأمثلة السابقة باستخدام البروتوكول *TCP (Transmission Control Protocol)* والذي جرى تصميمه لتحقيق الوثوقية وضمان وصول المعطيات المرسل.

هناك أيضاً بروتوكول آخر هو البروتوكول *UDP (User Datagram Protocol)*، وهو لا يضمن تسليم الرزم *packets* المرسل، كما أنه لا يضمن وصولها بالترتيب الذي أرسلت فيه. للوهلة الأولى قد يبدو لك هذا البروتوكول سيئاً، لكن الميزة الأساسية فيه هي أنه أسرع بكثير من البروتوكول السابق.

الدعم الذي تقدّمه جافا للبروتوكول UDP يشبه دعمها للبروتوكول TCP كثيراً ، إلا أنّه مع البروتوكول UDP عليك وضع المقبس DatagramSocket على الزبون والمخدّم سوياً. أيضاً هناك اختلاف آخر وهو أنّه لا داعي للقلق حول من يتكلم مع من وفي أيّ مكان بعد إجراء الاتصال. لكن يتوجّب على رزمة Datagram معرفة المكان الذي أتت منه والمكان الذي ستذهب إليه.

يقوم عنصر DatagramSocket بإرسال واستقبال الرزم. أما DatagramPacket فيحتوي على المعلومات، وعندما تقوم باستقبال datagram تحتاج فقط لدارئ buffer من أجل وضع المعطيات فيه. أما المعلومات التي تتعلق بعنوان إنترنت ورقم البوّابة فيتمّ تبديتها تلقائياً عندما تصل الرزمة من خلال DatagramSocket. لذلك فإنّ باني الصف DatagramPacket سيأخذ الشكل:

`DatagramPacket(buf, buf.length)`

وعندما يتم إرسال datagram، يجب أن يحتوي DatagramPacket على المعطيات، ليس هذا فقط وإنّما على عنوان الإنترنت ورقم البوّابة التي سترسل إليها. لذلك وفي هذه الحالة يأخذ باني الصف Datagram الشكل التالي:

`DatagramPacket(buf, length, inetAddress, port)`

حيث يحتوي buf على المعطيات المطلوب إرسالها، أما length فهو طول الدارئ buf، في حين يمثّل الوسيطان الأخيران عنوان الإنترنت inetAddress ورقم البوّابة port التي سيتم إرسال الرزمة إليها.

ومن أجل تسهيل إنشاء عنصر Datagram من سلسلة محارف String وبالعكس، يبدأ المثال التالي باستخدام أداة الصف Dgram:

```
//: Dgram.java
// A utility class to convert back and forth
// Between Strings and DataGramPackets.
import java.net.*;
public class Dgram {
    public static DatagramPacket toDatagram(
        String s, InetAddress destIA, int destPort)
    {
```

سلسلة الرضا للمعلومات

```
// Deprecated in Java 1.1, but it works:
byte[] buf = new byte[s.length() + 1];
s.getBytes(0, s.length(), buf, 0);
// The correct Java 1.1 approach, but it's
// Broken (it truncates the String):
// byte[] buf = s.getBytes();
return new DatagramPacket(buf, buf.length,
    destIA, destPort);
}
public static String toString(DatagramPacket
p){
    // The Java 1.0 approach:
    // return new String(p.getData(),
    // 0, 0, p.getLength());
    // The Java 1.1 approach:
    return
        new String(p.getData(), 0, p.getLength());
}
} ///:~
```

كما تلاحظ في هذا المثال فإن الطريقة الأولى toDatagram، والتي تأخذ الوسيط String و InetAddress و destport، تقوم بتوليد عنصر DatagramPacket، وذلك بنسخ محتوى سلسلة الحروف String ضمن الدائري byte وتمريره إلى باني DatagramPacket. أما الطريقة getByte() فتقوم بنسخ محارف السلسلة String ضمن الدائري byte.

والبرنامج التالي يوضح كيفية التعامل مع datagram ضمن المخدم:

```
//: ChatterServer.java
// A server that echoes datagrams
import java.net.*;
import java.io.*;
import java.util.*;
public class ChatterServer {
    static final int INPORT = 1711;
    private byte[] buf = new byte[1000];
    private DatagramPacket dp =
        new DatagramPacket(buf, buf.length);
```

```
// Can listen & send on the same socket:
private DatagramSocket socket;
public ChatterServer() {
    try {
        socket = new DatagramSocket(INPORT);
        System.out.println("Server started");
        while(true) {
            // Block until a datagram appears:
            socket.receive(dp);
            String rcvd = Dgram.toString(dp) +
                ", from address: " + dp.getAddress() +
                ", port: " + dp.getPort();
            System.out.println(rcvd);
            String echoString =
                "Echoed: " + rcvd;
            // Extract the address and port from the
            // received datagram to find out where to
            // send it back:
            DatagramPacket echo =
                Dgram.toDatagram(echoString,
                    dp.getAddress(), dp.getPort());
            socket.send(echo);
        }
    } catch(SocketException e) {
        System.err.println("Can't open socket");
        System.exit(1);
    } catch(IOException e) {
        System.err.println("Communication
            error");
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    new ChatterServer();
}
} ///:~
```

يحتوي العنصر `ChatterServer` على عنصر `DatagramSocket` وحيد لاستقبال الرسائل. و هو يمتلك على رقم بوابة الاستقبال فقط لوجوده على المخدم، لكن يجب أن يمتلك الزبون العنوان الصحيح للمكان الذي سيقوم بإرسال `datagram` إليه. أمّا داخل حلقة `while` اللانهائية، فيقوم العنصر `socket` بمخاطبة الطريقة `receive()`، ومن ثم يُجمّد حتى يصل `datagram`. يقوم بعد ذلك بوضعه في المستقبل المحدّد بالعنصر `DatagramPacket dp`، ثم يتم قلب الرزمة إلى سلسلة محارف `String` تحتوي على معلومات عن عنوان إنترنت وعن المقبس الذي أتت الرزمة منه.

من أجل اختبار هذا المخدم، سنقوم في البرنامج التالي بإنشاء عدّة مستخدمين يقومون جميعاً بإطلاق رزم `datagram` إلى المخدم وينتظرون أجوبتها:

```
//: ChatterClient.java
// Tests the ChatterServer by starting multiple
// clients, each of which sends datagrams.
import java.lang.Thread;
import java.net.*;
import java.io.*;
public class ChatterClient extends Thread {
    // Can listen & send on the same socket:
    private DatagramSocket s;
    private InetAddress hostAddress;
    private byte[] buf = new byte[1000];
    private DatagramPacket dp =
        new DatagramPacket(buf, buf.length);
    private int id;
    public ChatterClient(int identifier) {
        id = identifier;
        try {
            // Auto-assign port number:
            s = new DatagramSocket();
            hostAddress =
                InetAddress.getByName("localhost");
        } catch (UnknownHostException e) {
            System.err.println("Cannot find host");
        }
    }
}
```

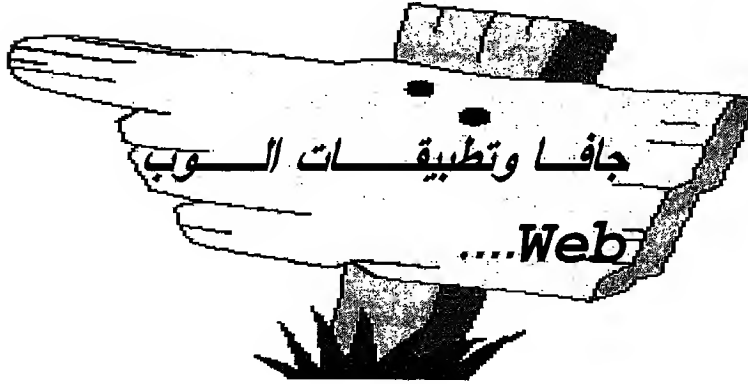
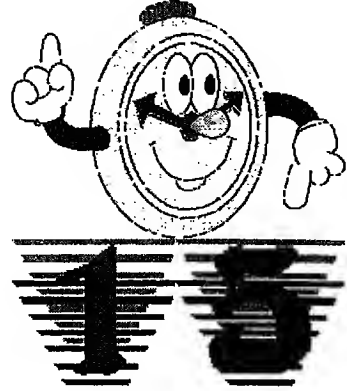
```

        System.exit(1);
    } catch(SocketException e) {
        System.err.println("Can't open socket");
        e.printStackTrace();
        System.exit(1);
    }
    System.out.println("ChatterClient
starting");
}
public void run() {
    try {
        for(int i = 0; i < 25; i++) {
            String outMessage = "Client #" +
                id + ", message #" + i;
            // Make and send a datagram:
            s.send(Dgram.toDatagram(outMessage,
                hostAddress,
                ChatterServer.INPORT));
            // Block until it echoes back:
            s.receive(dp);
            // Print out the echoed contents:
            String rcvd = "Client #" + id +
                ", rcvd from " +
                dp.getAddress() + ", " +
                dp.getPort() + ": " +
                Dgram.toString(dp);
            System.out.println(rcvd);
        }
    } catch(IOException e) {
        e.printStackTrace();
        System.exit(1);
    }
}
public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
        new ChatterClient(i).start();
}
} ///:~

```

في البرنامج السابق نلاحظ بأنه يجري إنشاء العنصر *ChatterClient* كنيسب *Thread* بحيث يمكن لعدة مستخدمين إزعاج المخدم. ويمكن هنا ملاحظة أن عنصر *DatagramPacket* الذي تم استقباله يشبه العنصر الذي استخدم في الصف *ChatterServer* إلى حد كبير. أما ضمن الباني، فيتم إنشاء عنصر *DatagramSocket* بدون وسطاء لأنه لا يتوجب عليه تحديد رقم بوابة خاص به. أما *hostAddress* فهو عبارة عن عنوان إنترنت للجهاز المضيف الذي ترغب بالتحدث معه. يجب أن يكون لهذا المضيف عنوان معروف ورقم بوابة محدد ليتمكن الزبائن من مخاطبته. ويُعطى كل نيسب *thread* رقم محدد وحيد *identification number*. ويتم إنشاء رسالة *String* ضمن الطريقة *run()* تحتوي على رقم محدد النيسب، ورقم الرسالة التي سيقوم هذا النيسب بإرسالها. تستخدم الرسالة *String* لإنشاء *datagram* الذي سيرسل إلى المضيف، حيث يتم أخذ رقم البوابة مباشرة الموجود ضمن ثابت تابع للصف *ChatterServer*. وبعد إرسال الرسالة، يتم تجميد الطريقة *receive()* حتى يقوم المخدم بإرجاع رسالة جواب.

يوضح المثال السابق أنه على الرغم من أن البروتوكول *UDP* غير موثوق، فإن جميع عناصر *datagrams* ستصل إلى المكان المطلوب.



في هذا الفصل بإنشاء تطبيق يعمل على الوب Web، وسنقوم من خلاله
سنقوم بإظهار جميع إمكانيات لغة جافا. جزء من هذا التطبيق عبارة عن برنامج
 جافا سيتم تنفيذه على مخدم وب Web server، أما الجزء الآخر فهو
 عبارة عن بريمج applet سيتم تثبيته على المستعرض browser. يقوم هذا
 البريمج بتجميع المعلومات من المستخدم وإرسالها إلى التطبيق العامل على مخدم الوب.

سلسلة الرضا للمعلومات

ستكون مهمة هذا البرنامج بسيطة: سيقوم أولاً البريمج بسؤال المستخدم عن عنوان بريده الإلكتروني *Email*، سيتحقق بعدها من صحة العنوان البريدي الذي تم إدخاله (سيتحقق من عدم وجود فراغات وسيتحقق من احتوائه على الرمز @)، ثم يقوم بإرسال هذا العنوان إلى مخدّم الويب *Web Server*. وسيقوم التطبيق العامل على المخدّم بالنقاط العنوان والتحقّق في ملف العناوين الموجود لديه، فإذا كان هذا العنوان موجوداً ضمن الملف سيرجع رسالة تدلّ على ذلك، ويتم إظهارها من قبل البريمج. أما إذا لم يكن هذا العنوان موجوداً ضمن ملف العناوين، فسيتم وضعه ضمن القائمة، وسيتم إخبار البريمج بأنّ عملية إضافة هذا العنوان تمتّ بنجاح.

الطريقة التقليدية لمعالجة هذا النوع من المشاكل تتم بإنشاء صفحة *HTML* مع حقل نصّي وزر تأكيد. ويمكن للمستخدم كتابة أيّ شيء ضمن الحقل النصّي وإرساله إلى المخدّم، وعند تأكيد قبول المعطيات المدخلة، سنقوم صفحة الويب بإخبار المخدّم عما سيفعله مع هذه المعطيات بتحديد برنامج واجهة البوابة المشتركة *CGI (Common Gateway Interface)* والذي سيقوم المخدّم بتشغيله بعد تلقيه هذه المعطيات.

ويكتب برنامج *CGI* هذا عادةً بلغة *Perl* أو *C* أو *C++*، وعليه معالجة كل شيء، فهو يقوم أولاً بفحص المعطيات للتأكد من أنّها بالشكل الصحيح، فإذا لم تكن كذلك يتوجّب على برنامج *CGI* إنشاء صفحة *HTML* لوصف المشكلة، حيث يتم ربطها بالمخدّم الذي يقوم بإرجاعها للمستخدم. ويجب على المستخدم تعديل الصفحة وإرسالها مجدداً. فإذا كانت المعطيات صحيحة، يقوم برنامج *CGI* بفتح ملف المعطيات وإضافة عنوان البريد الإلكتروني إليه أو البحث عنه إذا كان موجوداً في هذا الملف. وفي كلتا الحالتين يتوجّب عليه إنشاء صفحة *HTML* مناسبة للمخدّم من أجل إرسالها إلى المستخدم.

وكمبرمجين بلغة جافا، فإن الطريقة السابقة مناسبة لحلّ المشكلة، وسنحاول تنفيذ كل شيء بلغة جافا. لذلك سنقوم أولاً باستخدام بريمج جافا للتأكد من صحة المعطيات المدخلة في موقع الزبون، دون الحاجة إلى عمليات المرور المزعجة على الويب، ودون الحاجة لتنسيق الصفحات. ولنحاول بعد ذلك تجاوز مخدّم الويب، ولنقم بإنشاء اتصال شبكة خاص وذلك من البريمج إلى تطبيق جافا الموجود على المخدّم.

سنبدأ أولاً بإنشاء تطبيق المخدم...

سنقوم الآن بإنشاء تطبيق مخدم اسمه *NameCollector*، فإذا وجد أكثر من مستخدم يحاول تأكيد إدخال عنوان بريده الإلكتروني في نفس الوقت، سيقوم *NameCollector* باستخدام مقابس *TCP/IP*، ومن ثم استخدام تقنية تعدد النياسب *multithreading* التي تم شرحها في الفصول السابقة، من أجل التعامل مع أكثر من مستخدم في نفس الوقت.

لكن وعلى اعتبار أن أكثر من نيسب *thread* سيحاولون الكتابة على الملف الذي يحتوي على جميع عناوين البريد الإلكتروني، سنحتاج عندها إلى استخدام تقنية قفل *lock michanism* للتحقق من عدم وصول أكثر من نيسب واحد إلى الملف في وقت معين.

لكن ثبت بأن نسخة *Java 1.0* لا تستطيع معالجة ملف العناوين بسهولة (بعكس *Java 1.1*)، لذلك سنقوم بحل هذه المشكلة من خلال كتابة برنامج بلغة *C*، وسيفيدنا ذلك أيضاً بتوضيح كيفية ربط برنامج مكتوب بلغة غير لغة جافا مع برنامج مكتوب بلغة جافا.

ويمتلك عنصر *Runtime* طريقة اسمها *exec()* تقوم باستنهاض برنامج آخر على الجهاز وإرجاع عنصر *Procces*. وتستطيع الحصول على عنصر *OutputStream* يقوم بالاتصال مع الدخل القياسي من أجل هذا البرنامج المنفصل وكذلك عنصر *InputStream* يقوم بالاتصال مع الخرج القياسي. كل مايتوجب عليك عمله هو كتابة برنامج بأي لغة، يحصل على دخله من الدخل القياسي و يكتب خرجه على الخرج القياسي. وهو حل مناسب عندما تحتاج إلى حل مشكلة قد تكون معقدة بلغة جافا.

في البرنامج التالي المكتوب بلغة *C* (لأن جافا غير مناسبة لبرمجة *CGI*) سنوضح كيفية إدارة قائمة عناوين البريد الإلكتروني. سيقوم الدخل القياسي بقبول عنوان البريد الإلكتروني والبحث عن وجوده في القائمة، حيث سيقوم بإضافته إلى هذه القائمة في حال عدم إيجادها.

```
//: Listmgr.c
```

سلسلة الرضا للمعلومات

```
// Used by NameCollector.java to manage
// the email list file on the server
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BSIZE 250
int alreadyInList(FILE* list, char* name) {
    char lbuf[BSIZE];
    // Go to the beginning of the list:
    fseek(list, 0, SEEK_SET);
    // Read each line in the list:
    while(fgets(lbuf, BSIZE, list)) {
        // Strip off the newline:
        char * newline = strchr(lbuf, '\n');
        if(newline != 0)
            *newline = '\0';
        if(strcmp(lbuf, name) == 0)
            return 1;
    }
    return 0;
}
int main() {
    char buf[BSIZE];
    FILE* list = fopen("emlist.txt", "a+t");
    if(list == 0) {
        perror("could not open emlist.txt");
        exit(1);
    }
    while(1) {
        gets(buf); /* From stdin */
        if(alreadyInList(list, buf)) {
            printf("Already in list: %s", buf);
            fflush(stdout);
        }
        else {
            fseek(list, 0, SEEK_END);
            fprintf(list, "%s\n", buf);
            fflush(list);
        }
    }
}
```




```

printf("%s added to list", buf);
fflush(stdout);
}
}
} ///:~

```

تقوم الدالة الأولى *alreadyInList* في الملف بالتحقق من وجود الاسم المعطى كوسيط في الملف. هنا يتم تمرير اسم الملف كمؤشر إلى *FILE* وذلك إلى ملف مفتوح أصلاً داخل *main()*. أما الدالة *fseek()* فتقوم بالتنقل ضمن الملف حيث تم استخدامها هنا للتنقل إلى أعلى الملف. والدالة *fget()* تقوم بقراءة سطر من الملف *list* ووضعه في *lbuf*.

أما في البرنامج الرئيسي *main()* فيتم فتح الملف باستخدام *fopen()*. فإذا افترضنا أنه فتح الملف قد تم بنجاح، سيدخل البرنامج في حلقة لانهائية حيث تقوم الدالة *gets(buf)* بقراءة سطر من الدخل القياسي (الذي سيتم ربطه ببرنامج جافا بالطبع)، ووضع هذا السطر في السدائ *buf*، والذي يتم تمريره إلى الدالة *alreadyInList()*.

فإذا لم يكن هذا السطر موجوداً في الملف، ستقوم الدالة *fseek()* بنقل المؤشر إلى نهاية الملف وكتابته هناك. وستقوم *printf()* بإعلامنا عن إضافة الاسم الجديد إلى القائمة.

سنقوم الآن بإنشاء برنامج بلغة جافا يبدأ أولاً باستدعاء برنامج *C* السابق لإجراء عملية الاتصال الضرورية. يقوم بعدها بإنشاء مقبس من نمط *datagram* وذلك من أجل الاستماع إلى حزم *datagram* الواردة من البرمج.

```

//: NameCollector.java
// Extracts email names from datagrams and
stores
// them inside a file, using Java 1.02.
import java.net.*;
import java.io.*;
import java.util.*;
public class NameCollector {
    final static int COLLECTOR_PORT = 8080;

```

```
final static int BUFFER_SIZE = 1000;
byte[] buf = new byte[BUFFER_SIZE];
DatagramPacket dp =
    new DatagramPacket(buf, buf.length);
// Can listen & send on the same socket:
DatagramSocket socket;
Process listmgr;
PrintStream nameList;
DataInputStream addResult;
public NameCollector() {
    try {
        listmgr =
            Runtime.getRuntime().exec("listmgr.exe");
        nameList = new PrintStream(
            new BufferedOutputStream(
                listmgr.getOutputStream()));
        addResult = new DataInputStream(
            new BufferedInputStream(
                listmgr.getInputStream()));
    } catch (IOException e) {
        System.err.println(
            "Cannot start listmgr.exe");
        System.exit(1);
    }
    try {
        socket =
            new DatagramSocket(COLLECTOR_PORT);
        System.out.println(
            "NameCollector Server started");
        while(true) {
            // Block until a datagram appears:
            socket.receive(dp);
            String rcvd = new String(dp.getData(),
                0, 0, dp.getLength());
            // Send to listmgr.exe standard input:
            nameList.println(rcvd.trim());
            nameList.flush();
        }
    }
}
```



```

byte[] resultBuf = new
byte[BUFFER_SIZE];
int byteCount =
    addResult.read(resultBuf);
if(byteCount != -1) {
    String result =
        new String(resultBuf, 0).trim();
    // Extract the address and port from
    // the received datagram to find out
    // where to send the reply:
    InetAddress senderAddress =
        dp.getAddress();
    int senderPort = dp.getPort();
    byte[] echoBuf = new
    byte[BUFFER_SIZE];
    result.getBytes(
        0, byteCount, echoBuf, 0);
    DatagramPacket echo =
        new DatagramPacket(
            echoBuf, echoBuf.length,
            senderAddress, senderPort);
    socket.send(echo);
}
else
    System.out.println(
        "Unexpected lack of result from " +
        "listmgr.exe");
}
} catch(SocketException e) {
    System.err.println("Can't open socket");
    System.exit(1);
} catch(IOException e) {
    System.err.println("Communication
    error");
    e.printStackTrace();
}
}
}
public static void main(String[] args) {

```

```
new NameCollector();
}
} ///:~
```

في بداية البرنامج نلاحظ اختيار البوابة أولاً، ومن ثم يتم إنشاء حزمة *Datagram* وتوليد مؤشر إلى *DatagramSocket*. أما التعريفات الثلاثة التالية فتتعلق بالاتصال مع برنامج *C*، حيث أن عنصر *Object* ينتج عند بدء استنهاض برنامج *C* من قبل برنامج جافا، ويقوم هذا العنصر بتوليد عناصر *InputStream* و *OutputStream* تمثل الخرج والدخل القياسي لبرنامج *C*، ويتم تغليف هذه العناصر مع دخل وخرج جافا القياسي: *PrintStream* و *DataInputStream*. العمل الأساسي لهذا البرنامج يتم داخل الباني *constructor*، فمن أجل بدء تنفيذ برنامج *C* يتم جلب عنصر *Runtime* الحالي، ويستخدم لاستدعاء الطريقة *exec()* التي تقوم بإرجاع العنصر *Process*. لاحظ أيضاً وجود العديد من الاستدعاءات البسيطة التي تؤدي إلى توليد دفق *streams* من *getOutputStream()* و *getInputStream()*. اعتباراً من هذه النقطة فإن كل ماتحتاجه هو إرسال المعطيات إلى الدفق *nameList* والحصول على النتائج من *addResult*. كما سبق، فإن عنصر *DatagramSocket* متصل ببوابة. وداخل حلقة *while* اللانهائية يقوم البرنامج باستدعاء طريقة *receive()* التي يتم تجميدها حتى يظهر *datagram*، حيث يتم وضع محتواه في *String rcvd*. يتم بعدها إلغاء الفراغات في نهاية كل سلسلة، وإرسالها إلى برنامج *C* في السطر:

```
nameList.println(rcvd.trim());
```

وهذا ممكن فقط لأن طريقة *exec()* في جافا تتيح الوصول إلى أي برنامج تنفذ يقوم بالقراءة من الدخل القياسي والكتابة في الخرج القياسي.

سنقوم بعد ذلك بإنشاء البريمج

...NameSender

سنقوم بكتابة هذا البريمج بنسخة Java 1.0 لكي نضمن تشغيله في أكبر عدد ممكن من المستعرضات، لذلك يفضل أن يكون عدد الصفوف الناتجة أصغرياً، فبدلاً من استخدام الصف Dgram سيتم وضع جميع عمليات Datagram على الخط online. إضافة إلى ذلك سيحتاج البريمج إلى نيسب Thread للاستماع إلى أجوبة المخدم، فبدلاً من جعل هذا النيسب منفصلاً، سيتم دمجهم ضمن البريمج بتنفيذ الواجهة Runnable. وبالطبع ليس من السهل قراءته إلا أنه سينتج لدينا برمج بصف وحيد:

```
//: NameSender.java
// An applet that sends an email address
// as a datagram, using Java 1.02.
import java.awt.*;
import java.applet.*;
import java.net.*;
import java.io.*;
public class NameSender extends Applet
    implements Runnable {
    private Thread pl = null;
    private Button send = new Button(
        "Add email address to mailing list");
    private TextField t = new TextField(
        "type your email address here", 40);
    private String str = new String();
    private Label
        l = new Label(), l2 = new Label();
    private DatagramSocket s;
    private InetAddress hostAddress;
    private byte[] buf =
        new byte[NameCollector.BUFFER_SIZE];
    private DatagramPacket dp =
        new DatagramPacket(buf, buf.length);
```

```
private int vcount = 0;
public void init() {
    setLayout(new BorderLayout());
    Panel p = new Panel();
    p.setLayout(new GridLayout(2, 1));
    p.add(t);
    p.add(send);
    add("North", p);
    Panel labels = new Panel();
    labels.setLayout(new GridLayout(2, 1));
    labels.add(l1);
    labels.add(l2);
    add("Center", labels);
    try {
        // Auto-assign port number:
        s = new DatagramSocket();
        hostAddress = InetAddress.getByName(
            getCodeBase().getHost());
    } catch (UnknownHostException e) {
        l1.setText("Cannot find host");
    } catch (SocketException e) {
        l1.setText("Can't open socket");
    }
    l1.setText("Ready to send your email
address");
}
public boolean action (Event evt, Object arg)
{
    if(evt.target.equals(send)) {
        if(pl != null) {
            // pl.stop(); Deprecated in Java 1.2
            Thread remove = pl;
            pl = null;
            remove.interrupt();
        }
        l2.setText("");
        // Check for errors in email name:
        str = t.getText().toLowerCase().trim();
    }
}
```



```

if(str.indexOf(' ') != -1) {
    l.setText("Spaces not allowed in name");
    return true;
}
if(str.indexOf(',') != -1) {
    l.setText("Commas not allowed in name");
    return true;
}
if(str.indexOf('@') == -1) {
    l.setText("Name must include '@'");
    l2.setText("");
    return true;
}
if(str.indexOf('@') == 0) {
    l.setText("Name must preceed '@'");
    l2.setText("");
    return true;
}
String end =
    str.substring(str.indexOf('@'));
if(end.indexOf('.') == -1) {
    l.setText("Portion after '@' must " +
        "have an extension, such as '.com'");
    l2.setText("");
    return true;
}
// Everything's OK, so send the name. Get a
// fresh buffer, so it's zeroed. For some
// reason you must use a fixed size rather
// than calculating the size dynamically:
byte[] sbuf =
    new byte[NameCollector.BUFFER_SIZE];
str.getBytes(0, str.length(), sbuf, 0);
DatagramPacket toSend =
    new DatagramPacket(
        sbuf, 100, hostAddress,
        NameCollector.COLLECTOR_PORT);
try {

```

```

        s.send(toSend);
    } catch (Exception e) {
        l.setText("Couldn't send datagram");
        return true;
    }
    l.setText("Sent: " + str);
    send.setLabel("Re-send");
    pl = new Thread(this);
    pl.start();
    l2.setText(
        "Waiting for verification " + ++vcount);
    }
    else return super.action(evt, arg);
    return true;
}
// The thread portion of the applet watches
for
// the reply to come back from the server:
public void run() {
    try {
        s.receive(dp);
    } catch (Exception e) {
        l2.setText("Couldn't receive datagram");
        return;
    }
    l2.setText(new String(dp.getData(),
        0, 0, dp.getLength()));
}
} ///:~

```

كما ترى فإن واجهة المستخدم *UI* الخاصة بالبرمجة بسيطة جداً. فالحقل النصي *TextField* يمكنك من كتابة عنوان بريدك الإلكتروني بداخله، والزر *Button* يساعدك على إرسال هذا العنوان إلى المخدم. كما توجد لصاقتان *Labels* لإعطاء معلومات الحالة للمستخدم.

واعتباراً من الآن يمكنك اعتبار العناصر *DatagramSocket* و *InetAddress* و *DatagramPacket* كمصائد لاتصالات الشبكة. يمكنك

أخيراً رؤية الطريقة (*run()*) التي تحتوي على تنفيذ جزء النيسب مما يساعد البريمج على الاستماع للأجوبة الواردة من المخدم.

تقوم الطريقة (*init()*) بتحديد القيم الابتدائية لواجهة المستخدم الرسومية *GUI* باستخدام أدوات الإظهار الاعتيادية، ثم تقوم بعدها بإنشاء عناصر *DatagramSocket* التي سيتم استخدامها لإرسال واستقبال الـ *Datagrams*.

الطريقة (*action()*) تراقب فقط إن ضغطت زر *send*، وفي حال قمت بضغط هذا الزر فإنها تتحقق من *Thread p1* لرؤية إن كان *null*، فإذا لم يكن كذلك فهذا يعني بأنه يوجد نيسب على قيد الحياة!!؟ وأول مرة يتم فيها إرسال الرسالة يتم إقلاع النيسب لمراقبة ظهور إجابة. لذلك إذا كان هناك نيسب عامل فهذا يعني بأنها ليست المرة الأولى التي يحاول فيها المستخدم إرسال الرسالة، حيث يتم وضع قيمة *null* في المؤشر *p1* ويتم مقاطعة المستمع *listener*. وبغض النظر عما إذا كانت هذه المرة الأولى التي تم فيها ضغط الزر فسيتم حذف النص في 12.

المجموعة التالية من التعليمات تتحقق من وجود أخطاء في أسماء البريد الإلكتروني *email*. وتستخدم الطريقة (*String.indexOf()*) للبحث عن المحارف التي لا تسمح بها وفي حال اكتشاف أحدها يتم إخبار المستخدم بذلك. لاحظ أن كل ذلك يتم بدون استخدام الشبكة مما يسرع من زمن التنفيذ كثيراً ويلغي مشكل الإنترنت.

وبعد أن يتم التحقق من الاسم، يتم حزمه في *Datagram* وإرساله إلى العنوان المضيف *host address* ورقم البوابة *port number* بنفس الطريقة التي تم شرحها في الفصل السابق. ويتم تغيير اللصاقة الأولى لإعلامك بأن عملية الإرسال قد تمت، كما يتم تغيير نص الزر بحيث يصبح *re-send*. عند هذه النقطة يتم إقلاع النيسب وتعلمك اللصاقة الثانية بأن البريمج ينتظر إجابة من المخدم.

وتستخدم طريقة النيسب (*run()*) العنصر *DatagramSocket* الموجود في *nameSender*، حيث تقوم الطريقة (*receive()*) بتجميده حتى تأتي حزمة *Datagram* من المخدم. أما الحزمة الناتجة فيتم وضعها في العنصر *DatagramPacket dp*. ويتم استحصا المعطيات من الحزمة ووضعها في

للصافقة الثانية في `NameSender`. عند هذه النقطة فإنّ هذا النيسب يصبح في عداد الأموات!! وإذا لم تأتِ إجابة ما من المخدم بعد وقت قليل، فقد لا يصبر المستخدم ويضغط الزر من جديد مما يؤدي إلى إنهاء النيسب الحالي. وعلى اعتبار أنّ النيسب يُستخدم للاستماع إلى إجابة ما فإنّ للمستخدم كامل الحرية باستخدام `UI`.

وماذا عن صفحة الوب؟

بالطبع فإنّه يتوجّب على البريمج الدخول إلى صفحة الوب. سنعطيك فيما يلي صفحة وب كاملة، يمكنها تلقائياً تجميع الأسماء من أجل توليد قائمة عناوين البريد الإلكتروني:

```
<HTML>
<HEAD>
<META CONTENT="text/html">
<TITLE>
Add Yourself to Bruce Eckel's Java Mailing List
</TITLE>
</HEAD>
<BODY LINK="#0000ff" VLINK="#800080"
BGCOLOR="#ffffff">
<FONT SIZE=6><P>
Add Yourself to Bruce Eckel's Java Mailing List
</P></FONT>
The applet on this page will automatically add
your email address to the
mailing list, so you will receive update
information about changes to the
online version of "Thinking in Java,"
notification when the book is in
print, information about upcoming Java
seminars, and notification about
the "Hands-on Java Seminar" Multimedia CD. Type
in your email address and
press the button to automatically add yourself
to this mailing list. <HR>
<applet code=NameSender width=400 height=100>
```



</applet>

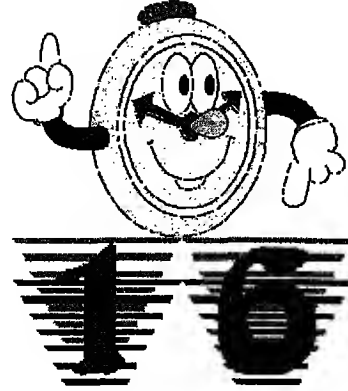
<HR>

If after several tries, you do not get verification it means that the Java application on the server is having problems. In this case, you can add yourself to the list by sending email to

Bruce@EckelObjects.com

</BODY>

</HTML>



وفقاً للإحصائيات والتوقعات التي تمّ إجراؤها في مجال تطوير البرمجيات، ثبت بأن أكثر من نصف عمليات التطوير خاصة بالعمليات المتعلقة بالمخدم/الزبون.

وتعتبر عملية الربط بين قواعد المعطيات المختلفة من أكبر المشاكل التي كانت تواجه مطوري تطبيقات قواعد المعطيات.

سلسلة الرضا للمعلومات

والقدرة على بناء منصة عمل *platform* مستقلة عن تطبيقات قواعد معطيات المخدم/الزبون، من أهم الأمور التي حاولت جافا الوصول إليها. وهذا ما أدى إلى بناء أداة ربط قواعد المعطيات *JDBC* (*Java Database Connectivity*) وذلك في الإصدار *Java 1.1*. ونظراً للأهمية التي أصبحت تتمتع بها لغة جافا، فلقد اعتمدت أنظمة إدارة قواعد المعطيات المعروفة هذه اللغة، خاصة بعد أن تكاملت هذه الأنظمة مع شبكة الإنترنت. وكانت شركة أوراكل *Oracle* السبّاقة في هذا المجال، حيث قامت بإنشاء حزمة برمجيات جديدة هي *Oracle AppBuilder for Java*، وذلك من أجل السماح للمطورين بإنشاء وتطوير برامج جافا بشكل سهل وسريع.

أداة الربط مع قواعد المعطيات *JDBC* ...

لقد تمّ تصميم *JDBC*، كأداة *API* في جافا، من أجل تسهيل وتبسيط الأمور. فاستدعاءات الطرق التي تطلبها تتوافق مع عمليات منطقية تقوم بها عندما تريد تجميع معلومات عن قاعدة المعطيات، هذه العمليات هي: الاتصال بقاعدة المعطيات، ثم إنشاء تعليمة الاستعلام وتنفيذها، وأخيراً النظر إلى المجموعة الناتجة.

وتقدّم *JDBC* أداة تسمى بمدير السوّاقّة *driver manager* تسمح ببناء منصة عمل مستقلة، حيث تقوم هذه الأداة، وبشكل تلقائي، بصيانة جميع عناصر السوّاقّات التي تحتاجها الاستعلامات في قاعدة معطياتك. لذلك إذا كان لديك ثلاثة أنواع من قواعد المعطيات التي ترغب بالاتصال معها، فستحتاج إلى ثلاثة عناصر سوّاقّات مختلفة.

وتقوم عناصر السوّاقّة بتسجيل نفسها مع مدير السوّاقّة *driver manager* ضمن وقت التحميل *loading*، حيث يمكن القيام بعملية التحميل باستخدام الطريقة *.Class.forName()*

من أجل فتح قاعدة معطيات، يتوجّب عليك إنشاء "database URL" تبيّن:
١. أنك قمت باستخدام *JDBC* مع "jdbc".

٢. البروتوكول الفرعي "subprotocol"، وهو اسم السوافة أو اسم تقنية وصل إلى قاعدة المعطيات. وعلى اعتبار أن تصميم JDBC مستوحى من تصميم ODBC، فإن البروتوكول الفرعي الأول المتاح هو "jdbc-odbc bridge" الموصوف بـ "odbc".

٣. محدد قاعدة المعطيات database identifier. ويتغير وفقا لسوافة قاعدة المعطيات المستخدمة، لكنه بشكل عام يزودنا باسم منطقي تمت مطابقته، من قبل برمجيات إدارة قاعدة المعطيات، مع الدليل الفيزيائي الذي تتوضع فيه جداول قاعدة المعطيات.

ويتم ضم جميع المعلومات السابقة في سلسلة محارف وحيدة تمثل database "URL". فمثلا للاتصال بقاعدة معطيات لها المحدد "people"، وذلك من خلال بروتوكول ODBC الفرعي، سيأخذ URL الخاص بقاعدة المعطيات الشكل:

```
String dbUrl = "jdbc:odbc:people";
```

وإذا قمت بالاتصال من خلال شبكة، سيحتوي URL قاعدة المعطيات على المعلومات التي تحدد الجهاز البعيد أيضا.

وعندما تكون جاهزا للاتصال بقاعدة المعطيات، قم باستدعاء الطريقة الساكنة `DriverManager.getConnection()` مع تمرير الوسيط التالية: URL الخاص بالقاعدة، واسم المستخدم، وكلمة المرور وذلك من أجل الدخول إلى قاعدة المعطيات. بالنتيجة ستحصل على عنصر `Connection` يمكنك استخدامه لاستعلام ومعالجة قاعدة المعطيات.

يقوم المثال التالي بفتح قاعدة معطيات معلومات الاتصال، ثم يبحث عن الاسم الأخير للشخص، كما هو معطى في سطر الأوامر. وسيقوم باختيار أسماء الأشخاص الذين يمتلكون عناوين بريد إلكتروني فقط، ثم يقوم بطباعة أسماء الأشخاص اللذين لهم نفس الاسم الأخير المعطى:

```
//: Lookup.java
// Looks up email addresses in a
// local database using JDBC
import java.sql.*;
public class Lookup {
```

سلسلة الرضا للمعلومات

```

public static void main(String[] args) {
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    try {
        // Load the driver (registers itself)
        Class.forName(
            "sun.jdbc.odbc.JdbcOdbcDriver");
        Connection c =
            DriverManager.getConnection(
                dbUrl, user, password);
        Statement s = c.createStatement();
        // SQL code:
        ResultSet r =
            s.executeQuery(
                "SELECT FIRST, LAST, EMAIL " +
                "FROM people.csv people " +
                "WHERE " +
                "(LAST='" + args[0] + "') " +
                "AND (EMAIL Is Not Null) " +
                "ORDER BY FIRST");
        while(r.next()) {
            // Capitalization doesn't matter:
            System.out.println(
                r.getString("Last") + ", "
                + r.getString("FIRST")
                + ": " + r.getString("EMAIL") );
        }
        s.close(); // Also closes ResultSet
    } catch (Exception e) {
        e.printStackTrace();
    }
}
} ///:~

```

من المثال السابق نلاحظ أنه بعد إجراء الاتصال من خلال
 DriverManager.getConnection()، تستطيع استخدام العنصر

Connection الناتج لإنشاء عنصر *Statement* باستخدام الطريقة *createStatement()*. ثم بعد ذلك يمكنك استدعاء الطريقة *executeQuery()* مع تمرير تعليمة *SQL* كوسيط. (سترى عما قريب أن بإمكانك توليد تعليمة *SQL* هذه تلقائياً، لذلك لن تكون بحاجة إلى معرفة لغة *SQL*).
أما الطريقة *executeQuery()* فتقوم بإرجاع عنصر *ResultSet*، وهو يشبه العداد قليلاً، حيث تقوم الطريقة *next()* بنقل العداد إلى التسجيل التالية في التعليمة، أو تقوم بإرجاع *null* عند الوصول إلى نهاية مجموعة النتيجة. وستحصل دوماً على عنصر *ResultSet* من الطريقة *executeQuery()* حتى لو كانت نتيجة الاستعلام مجموعة فارغة. ويجب عليك استدعاء الطريقة *next()* مرة واحدة قبل محاولة قراءة أي تسجيلية معطيات. فإذا كانت مجموعة النتيجة فارغة، فإن الاستدعاء الأول للطريقة *next()* سيقوم بإرجاع *false*. ومن أجل كل تسجيلية في مجموعة النتيجة يتم اختيار الحقول المطلوب إظهار قيمها.
عند هذه النقطة، ستحصل على معطيات قاعدتك بتنسيق جافا، حيث يصبح بإمكانك التعامل معها ضمن برامجك بسهولة.

أفضل توضيح ذلك بمثال عملي...

كما سترى فإن فهم الترميز باستخدام *JDBC* بسيط نسبياً. الأمر الأساسي الذي قد ترى فيه بعض الصعوبة يتجلى بإمكانية جعل هذا الترميز يعمل على نظامك الخاص، لأنه يتطلب منك تحديد سؤاقة *JDBC* وتحميلها بشكل صحيح.
بالطبع فإن الإجرائية السابقة تتغير بشكل كبير من جهاز إلى آخر، لكننا سنعمل هنا ضمن نظام *Windows 32-bit* كمثال لا أكثر.
الخطوات الأساسية في هذا المثال هي:

الخطوة الأولى: قم بإيجاد سؤاقة *JDBC*...

يحتوي البرنامج السابق على التعليمة:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

سلسلة الرضا للمعلومات

حيث يتم تحديد بنية الأدلة. وتظهر تعليمة التحميل السابقة من أجل سؤالات *jdbc* في بعض الأماكن فقط ضمن التوثيق الفعّال *online documentation*. وإذا لم تعمل معك تعليمة التحميل السابقة، فلربما يكون الاسم قد تغير بسبب تغير إصدار جافا، لذلك حاول البحث في جزء التوثيق ولا تيأس!!

الخطوة الثانية: توصيف القاعدة *Configure the Database*

...Database

هذه الخطوة خاصة بنظام *Windows 32-bit* أيضاً وقد تحتاج لإجراء بعض العمليات لتوصيف منصة العمل لديك. افتح أولاً لوحة التحكم *control panel*، افتح بعدها أيقونة *ODBC 32bit*، عليك هنا تحديد جزء التتويب *System DSN*، وكذلك جزء التتويب *File DSN*. يدعم *ODBC* القياسي العديد من أنماط الملفات كملفات *Dbase*، وملفات *ASCII* وغيرها.

فإذا أخذنا في مثالنا هنا القاعدة *people* وقمنا بتصديرها إلى ملف *ASCII*، فإن عملية التعامل مع هذا الملف ستكون بسيطة وسهلة. للقيام بذلك اختر *Add* من جزء التتويب *File DSN*، ثم اختر سؤافة النص التي يمكنها معالجة ملف *ASCII*. قم بإلغاء تحديد صندوق التحقق *use current directory* للسماح بتحديد الدليل الذي يحتوي على ملف المعطيات الذي تم تصديره. ستلاحظ بالطبع بأننا لم نتعامل مع ملف وحيد وإنما مع دليل، لأن قاعدة المعطيات تكون عادة ممثلة بمجموعة من الملفات ضمن دليل وحيد. ويحتوي كل ملف على جدول وحيد عادة، وعلى تعليمات *SQL* التي يمكنها توليد نتائج تم تجميعها من عدة جداول في قاعدة المعطيات.

الخطوة الثالثة: فحص التوصيف *Test The Configuration*

...Configuration

من أجل فحص التوصيف، ستحتاج إلى طريقة تمكنك من معرفة فيما إذا كانت قاعدة المعطيات مرئية من قبل البرنامج الذي يقوم بالاستعلام. يمكنك طبعا تنفيذ برنامج JDBC السابق وتضمين التعليمة:

```
Connection c = DriverManager.getConnection(
    dbUrl, user, password);
```

فإذا تم قذف استثناء، فإن هذا يدل على عدم صحة التوصيف.

وعلى أية حال فمن المفيد استخدام أداة توليد الاستعلام *query-generation tool* عند هذه النقطة. يمكنك مثلا استخدام الأداة *Microsoft Query* الموجودة ضمن *Microsoft Office*، لكن بإمكانك استخدام أدوات أخرى بالطبع. ستحتاج الأداة المستخدمة لمعرفة مكان قاعدة المعطيات، فمثلا مع *Microsoft Query*، يتوجب عليك الذهاب إلى جزء التكوين *File DSN* ضمن مدير *ODBC*، وإضافة مدخل جديد بتحديد سواقة النص *text driver*، والدليل الذي تقطن فيه قاعدة المعطيات.

الخطوة الرابعة: توليد استعلام SQL...

إن الاستعلام الذي تم إنشاؤه باستخدام الأداة *Microsoft Query* يقوم بإنشاء ترميز *SQL* بشكل تلقائي، وهنا علينا إدراج هذا الترميز ضمن برنامج جافا. نحتاج هنا إلى إنشاء استعلام يقوم بالبحث عن السجلات التي لها نفس الكنية *last name* والتي تم إدخالها ضمن سطر الأوامر عند بدء تنفيذ برنامج جافا. لنفترض مثلا أننا نبحث عن الكنية "*Noukari*" ونرغب بمعرفة عناوين البريد الإلكتروني لجميع الأشخاص الذين لهم هذه الكنية. من أجل ذلك سنتبع الخطوات التالية:

١. أنشئ استعلام جديد، واستخدم معالج الاستعلام *Query Wizard*، ثم اختر قاعدة المعطيات *people*.

٢. اختر من هذه القاعدة الجدول *people*، وحدد فيه الأعمدة

FIRST, LAST, EMAIL

٣. اختر LAST تحت Data Filter، ثم اختر equals مع الوسيط

"Noukari"، وانقر زر الراديو And.

٤. اختر EMAIL ثم اختر Is not Null.

٥. اختر أخيراً FIRST تحت Sort By.

يمكنك رؤية تعليمة SQL الموافقة للاستعلام السابق، بالنقر على زر SQL، وهذه التعليمة ستأخذ الشكل التالي:

```
SELECT people.FIRST, people.LAST, people.EMAIL
FROM people.csv people
WHERE (people.LAST='Eckel') AND
(people.EMAIL Is Not Null)
ORDER BY people.FIRST
```

الخطوة الخامسة: عدّل تعليمة SQL ثم الصقها...

كما تلاحظ في التعليمة السابقة، فإن أداة الاستعلام تقوم بتوليد التوصيف الكامل لجميع الأسماء، حتى لو كان لديك جدول وحيد. وفي هذه الحالة يمكنك إلغاء اسم الجدول من جميع أسماء الحقول لتصبح على الشكل:

```
SELECT FIRST, LAST, EMAIL
FROM people.csv people
WHERE (LAST='Eckel') AND
(EMAIL Is Not Null)
ORDER BY FIRST
```

إضافة إلى ذلك، فإنه بإمكانك البحث عن أي كنية وليس فقط الكنية "Noukari". قم فقط بتعديل التعليمة السابقة لتصبح على الشكل:

```
"SELECT FIRST, LAST, EMAIL " +
"FROM people.csv people " +
"WHERE " +
"(LAST='" + args[0] + "') " +
"AND (EMAIL Is Not Null) " +
"ORDER BY FIRST");
```

سنقوم بتوليد نسخة بواجهة مستخدم رسومية GUI

لبرنامج البحث السابق...

من المفيد جداً ترك برنامج البحث يعمل طوال الوقت، بحيث نستطيع الانتقال إليه في أية لحظة للبحث عن شخص ما. يقوم البرنامج التالي بإنشاء برنامج البحث كتطبيق/برمج:

```
application/applet:
//: VLookup.java
// GUI version of Lookup.java
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.sql.*;
public class VLookup extends Applet {
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    Statement s;
    TextField searchFor = new TextField(20);
    Label completion =
        new Label(" ");
    TextArea results = new TextArea(40, 20);
    public void init() {
        searchFor.addTextListener(new
            SearchForL());
        Panel p = new Panel();
        p.add(new Label("Last name to search
            for:"));
        p.add(searchFor);
        p.add(completion);
        setLayout(new BorderLayout());
        add(p, BorderLayout.NORTH);
        add(results, BorderLayout.CENTER);
        try {
            // Load the driver (registers itself)
            Class.forName(
```

```

"sun.jdbc.odbc.JdbcOdbcDriver");
    Connection c =
    DriverManager.getConnection(
        dbUrl, user, password);
    s = c.createStatement();
} catch (Exception e) {
    results.setText(e.getMessage());
}
}
}
class SearchForL implements TextListener {
    public void textValueChanged(TextEvent te)
    {
        ResultSet r;
        if(searchFor.getText().length() == 0) {
            completion.setText("");
            results.setText("");
            return;
        }
        try {
            // Name completion:
            r = s.executeQuery(
                "SELECT LAST FROM people.csv people "
                + "WHERE (LAST Like '" +
                searchFor.getText() +
                "%') ORDER BY LAST");
            if(r.next())
                completion.setText(
                    r.getString("last"));
            r = s.executeQuery(
                "SELECT FIRST, LAST, EMAIL " +
                "FROM people.csv people " +
                "WHERE (LAST='" +
                completion.getText() +
                "') AND (EMAIL Is Not Null) " +
                "ORDER BY FIRST");
        } catch (Exception e) {
            results.setText(
                searchFor.getText() + "\n");
        }
    }
}

```

```

        results.append(e.getMessage());
        return;
    }
    results.setText("");
    try {
        while(r.next()) {
            results.append(
                r.getString("Last") + ", "
                + r.getString("FIRST") +
                ": " + r.getString("EMAIL") + "\n");
        }
    } catch (Exception e) {
        results.setText(e.getMessage());
    }
}

}

public static void main(String[] args) {
    VLookup applet = new VLookup();
    Frame aFrame = new Frame("Email lookup");
    aFrame.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent
            e) {
                System.exit(0);
            }
        });
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(500,200);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
}
} ///:~

```

المثال السابق عبارة عن مثال تقليدي للتعامل مع قواعد المعطيات باستخدام واجهة المستخدم الرسومية GUI.

لاحظ هنا أنه أضيف العنصر *TextListener* للاستماع إلى *TextField*، بحيث يقوم بمحاولة استكمال الاسم، وذلك عن طريق البحث عنه في قاعدة المعطيات عندما تقوم بكتابة حرف جديد، وهو يقوم بإعطائك الاسم الأول الذي يقوم بإيجاده ممّا يساعدك كثيراً أثناء عملية البحث.

جافا وقواعد معطيات أوراكل...

كما ذكرنا سابقاً فلقد قامت شركة أوراكل بإضافة الأداة *Oracle AppBuilder* لبناء برامج جافا وتطويرها مع قواعد معطيات أوراكل. وهناك الكثير من المعالجات الموجودة في حزمة البرمجيات *Oracle AppBuilder for Java* وخاصةً *Data Frame Wizard* و *Deployment Wizard* التي تساعد على تحسين الإنتاجية إلى حد كبير عن طريق أتمتة الكثير من المهام المطلوبة، كما تساعد على تقصير الزمن اللازم لإنجاز هذه المهام، وذلك بتوليد الترميز الضروري لإنشاء عناصر جافا جديدة، وربط المكونات بقاعدة المعطيات.

أما معالج النشر *Deployment Wizard* فيساعد على إيجاد جميع صفوف جافا الضرورية لعمل المكونات *Component* بشكل سليم، ومن ثمّ أرشفتها في ملفات *ZIP* أو *JAR*. ويمكنك بعد ذلك إعادة استخدام أيّ مكون في أيّ من برمجيات أو تطبيقات جافا، وحتى أنه يمكنك تثبيتها على تطبيق المخدم للاستخدام الموزّع للشبكة.

إضافةً إلى ذلك يحتوي *Oracle AppBuilder for Java* على مترجم *compiler* من نمط *JIT (Just-In-Time)* للتطوير السريع، ولتحقيق الارتباطات الحبيبية الدقيقة *fine-grained dependency checker*. جميع التقنيات السابقة تقلّص زمن الترجمة في مرحلة تطوير المشروع، وتساعدك في الحصول على منتجك بوقتٍ أسرع بكثير.

سنقوم في الفقرات التالية بإعطائك فكرة عن تطوير التطبيقات باستخدام *AppBuilder*.

تطوير تطبيقات المخدم/الزبون Client/Server ...Application Development

يمكنك استخدام *AppBuilder* لبناء تطبيقات المخدم/الزبون التقليدية. والتطبيق مخدم/زبون، يعمل كبرنامج جافا على محطة عمل الزبون أو على حاسب شخصي، يمكنه الاتصال مباشرة مع مخدم معطيات *Oracle7* أو *Oracle8* أو *Oracle* *Lite* باستخدام سواقات *JDBC* التي يزودنا بها *AppBuilder*. يمكنك أيضا بناء بريمجات مخدم/زبون يتم تحميلها على مستعرض وب، ويمكن إيصالها مباشرة إلى مخدم معطيات أوراكل، باستخدام سواقة الزبون الرقيقة *JDBC* التي يزودنا بها *AppBuilder*. ولقد تم تنفيذ هذه السواقة الرقيقة بلغة جافا وبشكل كامل، لذلك يمكن تحميلها بسهولة، ولا تحتاج إلى تثبيت أي برمجيات إضافية من جانب الزبون. ويمكن تطوير تطبيقات أو بريمجات المخدم/الزبون دون استخدام أي من الخدمات *services* التي يزودنا بها مخدم تطبيقات أوراكل *Oracle Application Server*. ومع *AppBuilder* بإمكانك استخدام عدة طرق لتطوير بريمجات أو تطبيقات المخدم/الزبون:

✓ *JBCL*: أو ما يسمى *Java Beans Component Library* والتي تسمح لك باستخدام مكونات مجموعة المعطيات الفعالة. وهي تساعدك على أتمتة معالجة الاستعلامات *query processing* وتعديلات قواعد المعطيات *database updates* باستخدام تصميم مرئي *visual design*. ويمكن لمكونات مجموعة المعطيات *dataset components* ربط عمليات قاعدة المعطيات بتحكمات واجهة المستخدم *UI Controls*، كالشبكات *grids* وصناديق القوائم *list boxes* ومناطق النص *text areas*.

سلسلة الرضا للمعلومات

✓ *SQLJ*: يمكنك استخدام *Oracle SQLJ precompiler* عندما تقوم بتطوير تطبيقات ساكنة تماما *purely static applications* (وذلك في حال معرفتك لجداول وأعمدة *SQL* المستخدمة عند الترجمة). وتسمح لك *SQLJ* بالترميز و بمستوى أعلى من *JDBC* عن طريق تضمين تعليمات *SQL* مباشرة داخل ترميز جافا. ويقوم *SQLJ precompiler*، المكامل في *AppBuilder*، بترجمة تعليمات *SQL* إلى تعليمات جافا إضافة إلى ترميز *JDBC*. لذلك تسمح لك الأداة *SQLJ* مع *AppBuilder* بكتابة وتدقيق التطبيقات بشكل أسرع بكثير من استخدام *JDBC* فقط.

✓ *JDBC*: يمكنك كتابة ترميز تطبيقاتك بلغة جافا بشكل كامل مع استخدام *JDBC API* وذلك عندما تحتاج إلى بناء تحكم حبيبي دقيق *fine-grained control* للنفاذ إلى قاعدة المعطيات، أو عندما تقوم بتطوير تطبيق يحتاج إلى معلومات دقيقة حول *database metadata*. فعلى سبيل المثال، تسمح لك الأداة *JDBC* بالضبط الدقيق لعملية النفاذ إلى قاعدة المعطيات عن طريق جلب العديد من الأسطر ودفع التعديلات.

ملاحظة: يجب عليك التفريق بين الأداة *JDBC* وسوقاة *JDBC*. فجميع تطبيقات جافا، بغض النظر عن كيفية تطويرها أو مكان تنفيذها، تقوم بشكل أساسي باستخدام سوقات للاتصال بأوراكل. أما الترميز باستخدام *JDBC API* الصافي فهو عبارة عن تطوير بمستوى منخفض *low-level development* مشابه لاستخدام واجهة استدعاء أوراكل (*Oracle Call Interface*) لتطوير تطبيق قاعدة معطيات.

ويمكنك المزج والاختيار بين الأدوات *JBCI* و *SQLJ* و *JDBC* لتطوير تطبيقات المخدم/الزبون. فعلى سبيل المثال يمكنك بشكل مبني استخدام مكونات *JBCI* لتسهيل التطوير، لكن باستطاعتك في نفس التطبيق ترميز طلبات *JDBC* للاستخدامات المخصصة.

يمكن أيضا استخدام الأدوات *SQL* و *JDBC* لتطوير تطبيقات الطبقات المتعددة *multi-tier applications* وتطبيقات الويب *web-based applications*.

تطوير التطبيقات متعددة الطبقات Multi-Tier

...Application Development

هناك العديد من الطرق التي تساعدك على تطوير تطبيقات الويب. ويتألف النموذج الأساسي لتطبيق وب من أخطوطات CGI scripts (Common Gateway Interface)، مكتوبة لتوصيف CGI باستخدام لغة كلغة Perl مثلا. وأخطوطات CGI سهلة التطوير، ويمكنها خدمة النماذج البسيطة، والتي لا يتم استدعاؤها من قبل عدة مستخدمين بشكل جيد. لكن هناك بعض المشاكل العامة مع تطبيقات CGI، فهذه التطبيقات مقيدة بالعمل على معالج وحيد. إضافة إلى ذلك، فكل طلب CGI يعني بأنه يجب إغلاق إجرائية process جديدة لمعالجة هذا الطلب.

باستطاعتك إنجاز حلول قابلة للفهم والقياس باستخدام AppBuilder مع Oracle Application Server لتطوير تطبيقات الطبقة الوسطى middle-tier applications وذلك بلغة جافا.

هناك تقنيتان أساسيتان لإنشاء هذا النوع من التطبيقات:

- ✓ إنشاء تطبيقات الويب باستخدام خرطوشة Jweb.
- ✓ استخدام مكونات Java CORBA.

وتزودك الأداة AppBuilder بالعديد من المعالجات التي يمكنك استخدامها للبدء بإنشاء تطبيقات موزعة Distributed Applications. فهناك على سبيل المثال معالج يساعدك في توليد صفوف تغليف جافا بشكل سهل جدا، مما يسمح لتطبيقات الطبقة الوسطى باستدعاء إجراءات PL/SQL المخزنة في مخزن معطيات أوراكل. وهناك أيضا المعالج AppBuilder Deployment الذي يسمح لك بنشر التطبيقات المعتمدة على مكونات في الطبقة الوسطى.

تطبيقات JWeb ...

يمكنك القيام بتطوير تطبيقات وب البسيطة جدا لكنها عالية الأداء وتنفيذها كصفحات HTML ضمن مستعرض ما، باستخدام HTML المولد تلقائيا. وبزودنا مخدم تطبيقات أوراكل OAS بخروطوشة JWeb التي تعتبر بيئة تنفيذ لتطبيقات جافا وذلك في جهة المخدم. كما يزودنا مخدم تطبيقات أوراكل OAS بمجموعة أدوات وب Web Toolkit، وهي عبارة عن مجموعة من صفوف جافا يمكنها توليد HTML، والنفاذ إلى معطيات المخدمات، وجلب وتحديد ترويسات HTTP، والنفاذ إلى خدمات Web Request Broker.

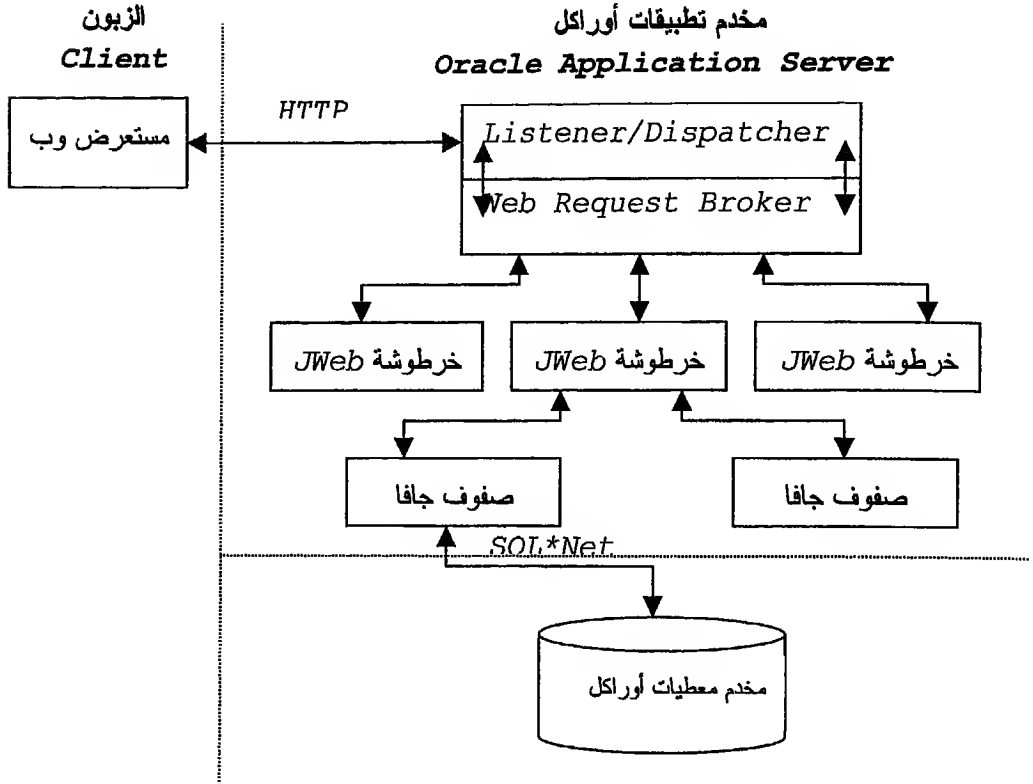
وباستطاعتك استخدام المعالج HTML-Java Wizard من أجل تطوير تطبيقات JWeb Cartridge.

قم بتضمين ملف HTML، ضمن مشروع AppBuilder يحتوي على HTML ساكن وعلى علامات <WRB_INC> خاصة تحدد موقع HTML المولد بشكل ديناميكي. بعدها يقوم المعالج بتوليد صف جافا يحتوي على طريقة لكل علامة <WRB_INC> في ملف HTML. ويمتلك ملف java الذي يقوم المعالج بتوليده نفس اسم ملف HTML. ويتم استكمال هذه الطريقة بترميز جافا الذي يقوم بتوليد HTML.

وعندما يستدعي مستعرض الوب صفحة HTML، يقوم مخدم تطبيقات أوراكل OAS بمعالجة هذا الطلب، حيث يقوم بدمج HTML الساكن مع HTML الديناميكي الناتجين عن خرطوشة JWeb Cartridge، ويعيد الصفحة النهائية إلى المستعرض.

هذه الطريقة أفضل من استخدام أخطوطات CGI، والسبب هو أن كل خرطوشة جافا Java Cartridge تحتفظ بآلة جافا الوهمية JVM (Java Virtual Machine) الخاصة بها، والتي يمكن تشغيلها من قبل مخدم التطبيق، مما يزيد من أداء الاستجابة للطلبات الواردة. كما أن خرطوشة جافا تستخدم تسهيلات الأداة WRB (Web Request Broker) والتي تساعد على تحميل التطبيق بشكل متوازن.

وأثناء وقت التنفيذ، تتفاعل صفحة HTML مع صفوف تطبيق جافا وقاعدة المعطيات كما في الشكل التالي:



تتلقى الطرق الموجودة في الخرطوشة المعطيات من مخدم قاعدة المعطيات، ثم تقوم بتوليد HTML اعتمادا على هذه المعطيات. يتم بعد ذلك إرجاع ملف HTML إلى صفحة المستعرض باستخدام البروتوكول HTTP من أجل إظهار المعطيات كنص.

النفاز إلى إجراءات قاعدة المعطيات المخزنة Accessing Database Stored ...Procedures

تمتلك خرطوشة جافا *OAS Java cartridge* أداة تسمح لك بإنشاء صفوف للنفاذ إلى إجراءات *PL/SQL* المخزنة، وإلى دالات قاعدة المعطيات. هذه الأداة هي *pl2java* التي تسمح لك بالاتصال مع مخطط قاعدة المعطيات، وإنشاء صف لكل حزمة *PL/SQL* في هذا المخطط.

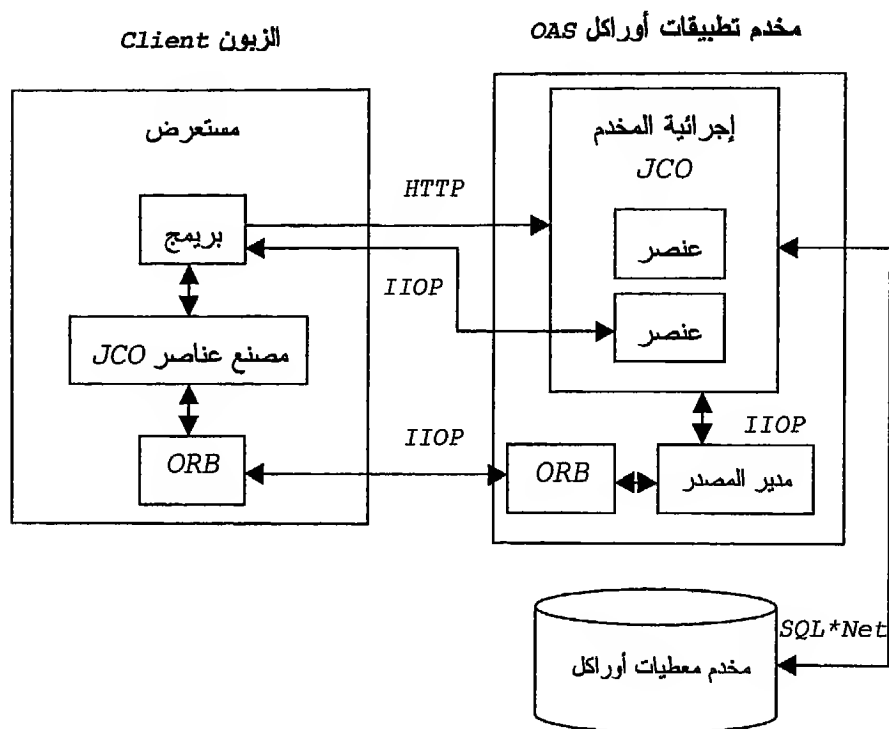
تطبيقات JCORBA ...

يدعم مخدم تطبيقات أوراكل *OAS* مكون نموذج التطبيق *application model* وذلك من خلال الخرطوشة *JCORBA*. وباستخدام هذا النموذج لن تصبح مقيدا بإجراء الاستدعاء من مستعرض وب، بل و يمكنك تطوير برنامج زبون *client* *program* يعمل كبرمج *applet* ضمن المستعرض، وبإمكانك أيضا كتابة برنامج زبون يعمل كتطبيق جافا.

يمكنك استخدام *AppBuilder* لكتابة واختبار عناصر *JCORBA* التي ستعمل على مخدم التطبيق. وتتم جميع الاتصالات بين الزبون والتطبيق الموزع *distributed application*، وكذلك بين عناصر *JCORBA* المختلفة من خلال *IIOP*. وبعد أن تتم عملية التبدئة *initialization* لن تتمكن طلبات الزبون وإجابات التطبيق من المرور من خلال مخدم التطبيق.

وبإمكانك توزيع تطبيقك لتنفيذه على عدة عقد في الشبكة. ويقوم مخدم التطبيق بتوليد هيكل وجسم برنامج *CORBA* المطلوب، لذلك لن تكون بحاجة لفهم *CORBA* من أجل تنفيذ ونشر تطبيق *CORBA*.

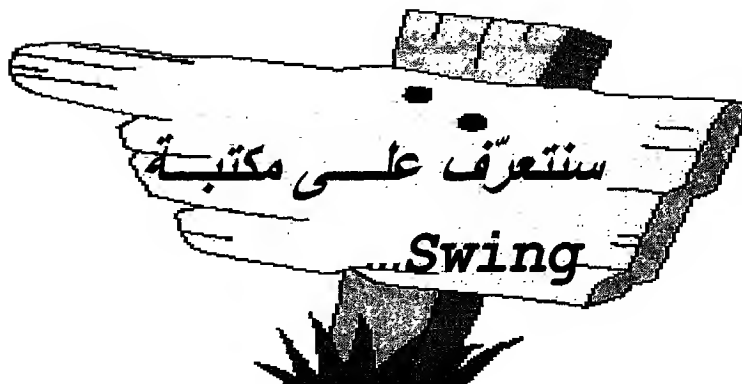
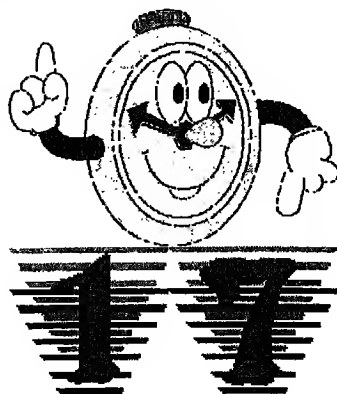
وبعد أن نقوم بتنفيذ تطبيق CORBA سنتمكن من النفاذ إلى مدير المخدم Server Manager، الذي سيساعدك على توصيف وإقلاع وإيقاف التطبيق. يوضح الشكل التالي علاقة الارتباط بين زبون JCO (وهو هنا بريمج)، وتطبيق يعتمد على JCO.



يبدأ التطبيق بالعمل عندما يطلب المستعرض صفحة HTML تحتوي على بريمج جافا. يقوم هنا مخدم التطبيق بشحن الصفحة، وملفات صف البرمج، وملف JCO JAR إلى الزبون.

يبدأ البرمج بالعمل، حيث يقوم بطلب إنشاء عنصر خاص في تطبيق JCO. عندها يبدأ مصنع عناصر JCO بالتفاعل مع ORB الموجود في المستعرض، وORB الموجود في

مخدم التطبيق للحصول على مرجع عنصر *object reference*. وتتم جميع اتصالات ORB عن طريق استخدام IIOP. يقوم بعدها مدير المصدر *Resource Manager* بإنشاء ممثل *instance* للعنصر المحدد ضمن إجرائية المخدم JCO. وتتم إعادة مرجع العنصر *object reference* إلى البريمج عن طريق الرجوع الموضح في المخطط السابق. يقوم البريمج بعدها باستخدام مرجع العنصر المعاد لاستدعاء الطرق على العنصر مباشرة. وباستخدام *AppBuilder* يمكنك توليد ملفات *JCO.APP* وملفات الأرشفة *jar*. الضرورية لنشر تطبيق JCORBA ضمن مخدم التطبيق.



مكتبة Swing بعد ظهور الإصدار 1.1 Java وظهور مكتبة
ظهرت AWT الجديدة فيه. لذلك يمكن اعتبار مكتبة Swing جزءاً من الإصدار
 Java 1.2، و التي يمكنك استخدامها مع الإصدار 1.1 Java.

تحتوي مكتبة *swing* على جميع المكونات التي افتقدتها في الإصدارات القديمة من جافا، لذلك يمكنك أن تتوقع رؤية العديد من الأدوات الرسومية الحديثة، كالأزرار التي تحتوي على صور والأشجار *tree* والجداول *tables* وغيرها. وتعتبر مكتبة *Swing* من المكتبات الكبيرة، التي سنحاول إظهار قوتها وبساطتها. وعندما تبدأ باستخدام مكتبة *Swing*، ستجد بأنها خطوة هائلة نحو الأمام. وستجد بأنّ مكونات هذه المكتبة عبارة عن حبيبات *Beans*، لذلك يمكن استخدامها في أي بيئة تطوير تدعم الحبيبات *Beans*. كما أنّها تحتوي على مجموعة كاملة من مكونات واجهة المستخدم *UI*، ومحتويات هذه المكتبة كاملة مكتوبة بلغة جافا.

يمكنك قلب برامجك القديمة بسهولة...

إذا كنت تمتلك برامج قمت ببنائها ضمن الإصدار *Java 1.1*، فلن تضطر إلى رميها والاستغناء عنها، وإنما يمكنك قلبها إلى *Swing*. وفي أغلب الأحيان ليس عليك سوى وضع الحرف "J" في بداية أسماء الصفوف لمكونات *AWT* الموجودة لديك. يوضح المثال التالي كيفية قلب برنامج قديم إلى المكتبة الجديدة:

```
//: JButtonDemo.java
// Looks like Java 1.1 but with J's added
package cl7.swing;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import com.sun.java.swing.*;
public class JButtonDemo extends Applet {
    JButton
        b1 = new JButton("JButton 1"),
        b2 = new JButton("JButton 2");
    JTextField t = new JTextField(20);
    public void init() {
        ActionListener al = new ActionListener() {
```

```

public void actionPerformed(ActionEvent
e) {
    String name =
        ((JButton)e.getSource()).getText();
    t.setText(name + " Pressed");
}
};
b1.addActionListener(al);
add(b1);
b2.addActionListener(al);
add(b2);
add(t);
}

public static void main(String args[]) {
    JButtonDemo applet = new JButtonDemo();
    JFrame frame = new JFrame("TextAreaNew");
    frame.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    frame.getContentPane().add(
        applet, BorderLayout.CENTER);
    frame.setSize(300,100);
    applet.init();
    applet.start();
    frame.setVisible(true);
}
} ///:~

```

كما تلاحظ فلقد أضيفت تعليمة `import` جديدة، وماعدا ذلك فإن كل شيء يشبه مكتبة `Java 1.1 AWT` مع إضافة حرف "J" إلى بداية أسماء الصفوف. ولن نقوم باستخدام الطريقة `add()` لإضافة أي شيء إلى الصف `Jframe`، وإنما يتوجب عليك جلب محتوى اللوحة أولاً. وبسبب تعليمة `package`، يتوجب عليك تنفيذ هذا البرنامج بكتابة:

 java c17.swing.JbuttonDemo

إظهار إطار عمل *framework* ...

على الرغم من أن البرامج المؤلفة من برمجيات وتطبيقات تكون قيمة، إلا أنه يجب الانتباه إلى كيفية استخدامها. وبدلاً من ذلك يمكن استخدام إطارات الإظهار *framework* بشكل فعال، لذلك سنقوم باستخدامها في بقية أمثلة هذا الفصل:

```
//: Show.java
// Tool for displaying Swing demos
package c17.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
public class Show {
    public static void
    inFrame(JPanel jp, int width, int height) {
        String title = jp.getClass().toString();
        // Remove the word "class":
        if(title.indexOf("class") != -1)
            title = title.substring(6);
        JFrame frame = new JFrame(title);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        frame.getContentPane().add(
            jp, BorderLayout.CENTER);
        frame.setSize(width, height);
        frame.setVisible(true);
    }
} ///:~
```



ويجب توريث الصفوف التي سيتم إظهارها من الصف *Jpanel*، وإضافة أية مكونات مرئية إليها. وفي النهاية يتم إنشاء الطريقة *main()* التي تحتوي على السطر:

```
Show.inFrame(new MyClass(), 500, 300);
```

صناديق إيضاح الأدوات *Tool tips*...

أغلب الصفوف التي ستحتاجها لإنشاء الواجهات الخاصة بك مشتقة من الصف *Jcomponent* الذي يحتوي على طريقة بالاسم *setToolTipText(String)*. الآن لتوصيف أي شيء بشكل مؤقت على نموذجك يمكنك كمثال كتابة:

```
jc.setToolTipText("My tip");
```

وعندما تبقى الفأرة لفترة من الوقت فوق العنصر *Jcomponent*، يظهر صندوق صغير يحتوي على نص بجانب الفأرة.

الإطارات *Borders*...

يحتوي الصف *Jcomponent* على طريقة بالاسم *setBorder()*. تسمح لك هذه الطريقة بوضع إطارات مختلفة على أي مكون مرئي. وفي المثال التالي سنقوم بتوضيح كيفية استخدام إطارات مختلفة عن طريق *showBorder()* التي تقوم بإنشاء عنصر *JPanel*، ثم تقوم باستخدام *RTTI* لإيجاد اسم الإطار الذي قمت باستخدامه، وتضع هذا الاسم في العنصر *JLabel* وسط اللوحة:

```
//: Borders.java
// Different Swing borders
package cl7.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
```

```
import com.sun.java.swing.border.*;
public class Borders extends JPanel {
    static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String nm = b.getClass().toString();
        nm = nm.substring(nm.lastIndexOf('.') + 1);
        jp.add(new JLabel(nm, JLabel.CENTER),
            BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
    }
    public Borders() {
        setLayout(new GridLayout(2,4));
        add(showBorder(new TitledBorder("Title")));
        add(showBorder(new EtchedBorder()));
        add(showBorder(new
            LineBorder(Color.blue)));
        add(showBorder(
            new MatteBorder(5,5,30,30,Color.green)));
        add(showBorder(
            new BevelBorder(BevelBorder.RAISED)));
        add(showBorder(
            new SoftBevelBorder(BevelBorder.LOWERED)));
        add(showBorder(new CompoundBorder(
            new EtchedBorder(),
            new LineBorder(Color.red))));
    }
    public static void main(String args[]) {
        Show.inFrame(new Borders(), 500, 300);
    }
} ///:~
```

الأزرار ...Buttons

تحتوي مكتبة Swing على أنماط مختلفة من الأزرار، ولقد قامت بتغيير تنظيم العناصر، حيث يتم توريث جميع الأزرار وصناديق التحقق وأزرار الراديو وحتى عناصر القوائم من الصف `AbstractButton`.

سنقوم في هذا المثال بإظهار أنماط الأزرار المتاحة:

```
//: Buttons.java
// Various Swing buttons
package cl7.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.basic.*;
public class Buttons extends JPanel {
    JButton jb = new JButton("JButton");
    BasicArrowButton
        up = new BasicArrowButton(
            BasicArrowButton.NORTH),
        down = new BasicArrowButton(
            BasicArrowButton.SOUTH),
        right = new BasicArrowButton(
            BasicArrowButton.EAST),
        left = new BasicArrowButton(
            BasicArrowButton.WEST);
    Spinner spin = new Spinner(47, "");
    StringSpinner stringSpin =
        new StringSpinner(3, "",
            new String[] {
                "red", "green", "blue", "yellow" });
    public Buttons() {
        add(jb);
        add(new JToggleButton("JToggleButton"));
        add(new JCheckBox("JCheckBox"));
        add(new JRadioButton("JRadioButton"));
        up.addActionListener(new ActionListener() {
```

```

public void actionPerformed(ActionEvent
e) {
    spin.setValue(spin.getValue() + 1);
}
});
down.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent
e) {
        spin.setValue(spin.getValue() - 1);
    }
});
JPanel jp = new JPanel();
jp.add(spin);
jp.add(up);
jp.add(down);
jp.setBorder(new TitledBorder("Spinner"));
add(jp);
left.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent
e) {
        stringSpin.setValue(
            stringSpin.getValue() + 1);
    }
});
right.addActionListener(new
ActionListener() {
    public void actionPerformed(ActionEvent
e) {
        stringSpin.setValue(
            stringSpin.getValue() - 1);
    }
});
jp = new JPanel();
jp.add(stringSpin);
jp.add(left);
jp.add(right);

```



```

jp.setBorder(
    new TitledBorder("StringSpinner"));
add(jp);
}
public static void main(String args[]) {
    Show.inFrame(new Buttons(), 300, 200);
}
} ///:~

```

مجموعات الأزرار *Button Groups*...

لكي تستطيع إنشاء أزرار راديو، عليك إضافتها إلى مجموعة أزرار، وذلك بشكل مشابه لطريقة AWT القديمة، وإنما بطريقة أفضل. وكما يوضح المثال التالي يمكن إضافة أي زر *AbstractButton* إلى مجموعة أزرار *ButtonGroup*. ولتجنب تكرار الكثير من التراميز، يقوم هذا المثال باستخدام تقنية الانعكاس *reflection* لتوليد مجموعة من الأزرار المختلفة، وهذا واضح في الصف *makeBPanel* الذي يقوم بإنشاء مجموعة أزرار مختلفة:

```

//: ButtonGroups.java
// Uses reflection to create groups of
different
// types of AbstractButton.
package cl7.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.border.*;
import java.lang.reflect.*;
public class ButtonGroups extends JPanel {
    static String[] ids = {
        "June", "Ward", "Beaver",
        "Wally", "Eddie", "Lumpy",
    };
    static JPanel

```

```

makeBPanel(Class bClass, String[] ids) {
    ButtonGroup bg = new ButtonGroup();
    JPanel jp = new JPanel();
    String title = bClass.getName();
    title = title.substring(
        title.lastIndexOf('.') + 1);
    jp.setBorder(new TitledBorder(title));
    for(int i = 0; i < ids.length; i++) {
        AbstractButton ab = new
            JButton("failed");
        try {
            // Get the dynamic constructor method
            // that takes a String argument:
            Constructor ctor =
                bClass.getConstructor(
                    new Class[] { String.class });
            // Create a new object:
            ab = (AbstractButton)ctor.newInstance(
                new Object[] {ids[i]});
        } catch(Exception ex) {
            System.out.println("can't create " +
                bClass);
        }
        bg.add(ab);
        jp.add(ab);
    }
    return jp;
}

public ButtonGroups() {
    add(makeBPanel(JButton.class, ids));
    add(makeBPanel(JToggleButton.class, ids));
    add(makeBPanel(JCheckBox.class, ids));
    add(makeBPanel(JRadioButton.class, ids));
}

public static void main(String args[]) {
    Show.inFrame(new ButtonGroups(), 500, 300);
}
} ///:~

```

كما تلاحظ يتم وضع اسم الصف في عنوان الإطار. وتتم تبديلة الصف *AbstractButton* بالعنصر *JButton* الذي يأخذ التسمية "Failed"، فإذا أهملت رسالة الاستثناء، ستظهر لك دائما نفس المشكلة على الشاشة. أما الطريقة *getConstructor()* فنقوم بتوليد عنصر *Constructor*، وكل ما عليك القيام به بعد ذلك هو استدعاء الطريقة *newInstance()* مع تمرير مصفوفة العناصر *Object* التي تحتوي على الوسطاء الحاليين.

الأيقونات Icons...

يمكنك استخدام أيقونة *Icon* داخل العنصر *JLabel* أو أي شيء يورث من الصف *AbstractButton* (وتتضمن *JButton* و *Jcheckbox* و *JradioButton* و *JmenuItem*). يوضح المثال التالي كيفية استخدام الأيقونات *Icons* مع الأزرار. يمكنك استخدام أي ملف *gif*، وفتح هذا الملف لجلب الصورة منه. قم فقط بإنشاء عنصر *ImageIcon* وربطه مع اسم الملف، تستطيع بعد ذلك استخدام الأيقونة الناتجة في برنامجك:

```
//: Faces.java
// Icon behavior in JButtons
package c17.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
public class Faces extends JPanel {
    static Icon[] faces = {
        new ImageIcon("face0.gif"),
        new ImageIcon("face1.gif"),
        new ImageIcon("face2.gif"),
        new ImageIcon("face3.gif"),
        new ImageIcon("face4.gif"),
    };
    JButton
```

```

jb = new JButton("JButton", faces[3]),
jb2 = new JButton("Disable");
boolean mad = false;
public Faces() {
    jb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent
        e){
            if(mad) {
                jb.setIcon(faces[3]);
                mad = false;
            } else {
                jb.setIcon(faces[0]);
                mad = true;
            }
            jb.setVerticalAlignment(JButton.TOP);
            jb.setHorizontalAlignment(JButton.LEFT);
        }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(faces[1]);
    jb.setPressedIcon(faces[2]);
    jb.setDisabledIcon(faces[4]);
    jb.setToolTipText("Yow!");
    add(jb);
    jb2.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent
        e){
            if(jb.isEnabled()) {
                jb.setEnabled(false);
                jb2.setText("Enable");
            } else {
                jb.setEnabled(true);
                jb2.setText("Disable");
            }
        }
    });
}

```

```

        add(jb2);
    }
    public static void main(String args[]) {
        Show.inFrame(new Faces(), 300, 200);
    }
} //:~

```

يمكنك استخدام أي أيقونة *Icon* في العديد من البانيات *constructors*، كما يمكنك استخدام الطريقة *setIcon()* لإضافة أيقونة *Icon* أو تغييرها. يوضح المثال السابق كيف يتمكن الصف *JButton* من تغيير نمط الأيقونة وفقا للعمل الذي يتم على الزر كالضغط أو عدم التفعيل وغير ذلك، مما يعطي للزر مشهدا أكثر إثارة.

القوائم *Menus*...

تحسنت القوائم وأصبحت أكثر مرونة باستخدام مكتبة *Swing*، فلقد أصبح بإمكانك استخدامها في أي مكان، حتى في اللوحات *panels* والبرمجيات *applets*. وتشبه طريقة إنشاء القوائم تلك التي استخدمت مع مكتبة *AWT* القديمة، مما يبقي على المشاكل التي كانت تواجهنا عند إنشاء قائمة، والتي تتمثل بالترميز القاسي المطلوب، وعدم وجود أية مصادر تدعم القوائم.

تقدم لنا الطريقة التالية خطوة نحو الأمام لحل المشاكل السابقة، وذلك بوضع جميع المعلومات المتعلقة بكل قائمة في مصفوفة ثنائية الأبعاد من *Object*. ويتم تنظيم هذه المصفوفة بحيث يحتوي السطر الأول على اسم القائمة، أما بقية أسطر المصفوفة فتحتوي على عناصر القائمة ومميزاتها.

```

//: Menus.java
// A menu-building system; also demonstrates
// icons in labels and menu items.
package cl7.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
public class Menus extends JPanel {

```

سلسلة الرضا للمعلومات

```
static final Boolean
    bT = new Boolean(true),
    bF = new Boolean(false);
// Dummy class to create type identifiers:
static class MType { MType(int i) {} };
static final MType
    mi = new MType(1), // Normal menu item
    cb = new MType(2), // Checkbox menu item
    rb = new MType(3); // Radio button menu
    item
JTextField t = new JTextField(10);
JLabel l = new JLabel("Icon Selected",
    Faces.faces[0], JLabel.CENTER);
ActionListener a1 = new ActionListener() {
public void actionPerformed(ActionEvent e) {
    t.setText(
        ((JMenuItem)e.getSource()).getText());
    }
};
ActionListener a2 = new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        JMenuItem mi = (JMenuItem)e.getSource();
        l.setText(mi.getText());
        l.setIcon(mi.getIcon());
    }
};
// Store menu data as "resources":
public Object[][] fileMenu = {
    // Menu name and accelerator:
    { "File", new Character('F') },
    // Name type accel listener enabled
    { "New", mi, new Character('N'), a1, bT },
    { "Open", mi, new Character('O'), a1, bT },
    { "Save", mi, new Character('S'), a1, bF },
    { "Save As", mi, new Character('A'), a1,
    bF},
    { null }, // Separator

```

```

    { "Exit", mi, new Character('x'), al, bT },
};

public Object[][] editMenu = {
    // Menu name:
    { "Edit", new Character('E') },
    // Name type accel listener enabled
    { "Cut", mi, new Character('t'), al, bT },
    { "Copy", mi, new Character('C'), al, bT },
    { "Paste", mi, new Character('P'), al, bT
    },
    { null }, // Separator
    { "Select All", mi, new
    Character('l'), al, bT },
};

public Object[][] helpMenu = {
    // Menu name:
    { "Help", new Character('H') },
    // Name type accel listener enabled
    { "Index", mi, new Character('I'), al, bT
    },
    { "Using help", mi, new
    Character('U'), al, bT },
    { null }, // Separator
    { "About", mi, new Character('t'), al, bT
    },
};

public Object[][] optionMenu = {
    // Menu name:
    { "Options", new Character('O') },
    // Name type accel listener enabled
    { "Option 1", cb, new Character('1'),
    al, bT },
    { "Option 2", cb, new Character('2'),
    al, bT },
};

public Object[][] faceMenu = {
    // Menu name:
    { "Faces", new Character('a') },

```

```
// Optinal last element is icon
{ "Face 0", rb, new Character('0'), a2, bT,
  Faces.faces[0] },
{ "Face 1", rb, new Character('1'), a2, bT,
  Faces.faces[1] },
{ "Face 2", rb, new Character('2'), a2, bT,
  Faces.faces[2] },
{ "Face 3", rb, new Character('3'), a2, bT,
  Faces.faces[3] },
{ "Face 4", rb, new Character('4'), a2, bT,
  Faces.faces[4] },
};

public Object[] menuBar = {
    fileMenu, editMenu, faceMenu,
    optionMenu, helpMenu,
};

static public JMenuBar
createMenuBar(Object[] menuBarData) {
    JMenuBar menuBar = new JMenuBar();
    for(int i = 0; i < menuBarData.length; i++)
        menuBar.add(
            createMenu((Object[][])menuBarData[i]));
    return menuBar;
}

static ButtonGroup bgroup;
static public JMenu
createMenu(Object[][] menuData) {
    JMenu menu = new JMenu();
    menu.setText((String)menuData[0][0]);
    menu.setKeyAccelerator(
        ((Character)menuData[0][1]).charValue());
    // Create redundantly, in case there are
    // any radio buttons:
    bgroup = new ButtonGroup();
    for(int i = 1; i < menuData.length; i++) {
        if(menuData[i][0] == null)
            menu.add(new JSeparator());
        else
```

```

        menu.add(createMenuItem(menuData[i]));
    }
    return menu;
}
static public JMenuItem
createMenuItem(Object[] data) {
    JMenuItem m = null;
    MType type = (MType)data[1];
    if(type == mi)
        m = new JMenuItem();
    else if(type == cb)
        m = new JCheckBoxMenuItem();
    else if(type == rb) {
        m = new JRadioButtonMenuItem();
        bgroup.add(m);
    }
    m.setText((String)data[0]);
    m.setKeyAccelerator(
        ((Character)data[2]).charValue());
    m.addActionListener(
        (ActionListener)data[3]);
    m.setEnabled(
        ((Boolean)data[4]).booleanValue());
    if(data.length == 6)
        m.setIcon((Icon)data[5]);
    return m;
}
Menus() {
    setLayout(new BorderLayout());
    add(createMenuBar(menuBar),
        BorderLayout.NORTH);
    JPanel p = new JPanel();
    p.setLayout(new BorderLayout());
    p.add(t, BorderLayout.NORTH);
    p.add(l, BorderLayout.CENTER);
    add(p, BorderLayout.CENTER);
}
public static void main(String args[]) {

```

```
Show.inFrame(new Menu(), 300, 200);
```

```
}  
} ///:~
```

الهدف من هذا المثال ببساطة السماح للمبرمج بإنشاء جداول تمثل كل قائمة، بدلا من كتابة أسطر ترميز لبناء هذه القوائم. كل جدول يولد قائمة، ويحتوي الجدول الأول على اسم القائمة ورمز مفتاح التسريع الموافق `keyboard accelerator`. أما بقية الأسطر فتحتوي على المعطيات المتعلقة بكل عنصر قائمة: السلسلة `string` المتوجب وضعها في عنصر القائمة، ونمط عنصر القائمة، ومسرّع لوحة المفاتيح `keyboard accelerator`، ومستمع الفعل `actionlistener` الذي يتم قدحه عند اختيار عنصر القائمة هذا، وأخيرا تحديد فيما إذا كان عنصر القائمة هذا فعالا أم لا. وعندما يبدأ السطر بكلمة `null` فتتم معاملته كفاصل `separator`.

ولتجنب عمليات إنشاء العناصر المنطقية `Boolean` وأعلام النمط `type flags` الكثيرة والمملة، فلقد تم إنشاؤها كقيم `static final` في بداية الصف: `bF` و `bT` لتمثيل القيم المنطقية `Boolean`، أما فتتمثل عناصر القوائم العادية `normal` `checkbox` `menu items`، و `cb` لعناصر قوائم صناديق التحقق `checkbox menu items`، و `rb` لعناصر قوائم أزرار الراديو `radio button menu items`.

يوضح هذا المثال أيضا كيف يمكن لصفوف `JLabels` و `JMenuItems` أن تحتوي على أيقونات `Icon`، حيث يتم وضع أيقونة `Icon` ضمن `JLabel` من خلال باني الصف، وتتغير هذه الأيقونة عند اختيار عنصر القائمة الموافق.

وتحتوي المصفوفة `menuBar` على مؤشرات إلى جميع قوائم الملف بالترتيب الذي ترغب به لإظهارها على شريط القائمة `menu bar`. ويتم تمرير هذه المصفوفة إلى الطريقة `(createMenuBar)` التي ستقوم بتقسيمها إلى مصفوفة منفردة من معطيات القائمة، حيث يتم تمرير كل منها إلى الطريقة `(createMenu)`. بدورها تقوم هذه الطريقة بأخذ السطر الأول من معطيات القائمة وإنشاء عنصر `JMenu` اعتمادا عليه، ثم تقوم باستدعاء الطريقة `(createMenuItem)` لكل سطر في بقية أسطر معطيات القائمة.

أخيرا نقوم الطريقة `() createMenuItem` بعبور كل سطر من أسطر معطيات القائمة وتحديد نمط القائمة وصفاتها، حيث نقوم بعد ذلك بإنشاء عنصر القائمة الموافق. وفي النهاية، وكما ترى في الباني `() Menus`، يتم إنشاء القائمة الموافقة للجدول السابقة بطلب `(menubar) createMenuBar` حيث تتم معالجة كل شيء بشكل متكرر.

القوائم المنبثقة *Popup Menus*...

هنالك صف خاص بهذا النوع من القوائم هو *JPopupMenu*، وقد يبدو التعامل معه غريبا بعض الشيء، حيث يتوجب عليك استدعاء الطريقة *enableEvent()* ثم اختيار حدث الفأرة بدلا من استخدام مستمع حدث *event listener*. ولو قممت باستخدام مستمع الحدث فلن يرجع حدث الفأرة *MouseEvent* القيمة *True* باستخدام الطريقة *isPopupTrigger()*.

يوضح المثال التالي كيفية إنشاء قائمة منبثقة:

```
//: Popup.java
// Creating popup menus with Swing
package cl7.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
public class Popup extends JPanel {
    JPopupMenu popup = new JPopupMenu();
    JTextField t = new JTextField(10);
    public Popup() {
        add(t);
        ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent
            e) {
                t.setText(
                    ((JMenuItem)e.getSource()).getText());
            }
        };
        JMenuItem m = new JMenuItem("Hither");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Yon");
        m.addActionListener(al);
        popup.add(m);
    }
}
```

```

    m = new JMenuItem("Afar");
    m.addActionListener(al);
    popup.add(m);
    popup.addSeparator();
    m = new JMenuItem("Stay Here");
    m.addActionListener(al);
    popup.add(m);
    enableEvents(AWTEvent.MOUSE_EVENT_MASK);
}
protected void processMouseEvent(MouseEvent
e) {
    if (e.isPopupTrigger())
        popup.show(
            e.getComponent(), e.getX(), e.getY());
    super.processMouseEvent(e);
}
public static void main(String args[]) {
    Show.inFrame(new Popup(), 200, 150);
}
} ///:~

```

ولقد تمت إضافة نفس العنصر `ActionListener` إلى كل عنصر قائمة `JMenuItem`، وهو يقوم ب جلب النص من عنوان القائمة وإدراجه في `JTextField`.

صناديق القائمة وصناديق السرد والتحرير

List boxes and combo

...boxes

تعمل هذه الصناديق في المكتبة الجديدة `Swing` كما في مكتبة `AWT` القديمة، لكن أصبح بإمكانك استخدام المزيد من الوظائف معها.

يوضح المثال التالي كيفية إنشاء صناديق القائمة وصناديق السرد والتحرير:

```
//: ListCombo.java
// List boxes & Combo boxes
package cl7.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
public class ListCombo extends JPanel {
    public ListCombo() {
        setLayout(new GridLayout(2,1));
        JList list = new JList(ButtonGroups.ids);
        add(new JScrollPane(list));
        JComboBox combo = new JComboBox();
        for(int i = 0; i < 100; i++)
            combo.addItem(Integer.toString(i));
        add(combo);
    }
    public static void main(String args[]) {
        Show.inFrame(new ListCombo(),200,200);
    }
} ///:~
```

الشيء الذي قد يثير استغرابك هو عدم تمكن *JLists* من إظهار شريط الانزلاق بشكل تلقائي. من أجل القيام بذلك تحتاج إلى تغليف عنصر *JList* في الصف *JScrollPane* وستتم إدارة جميع التفاصيل بعد ذلك بشكل تلقائي.

أشرطة التقدم *Sliders and ...progress bars*

تسمح لك الأداة *Slider* بإدخال المعطيات عن طريق تحريك نقطة معينة، وهي مفيدة في بعض الحالات (كأزرار التحكم بالصوت مثلاً). أما شريط التقدم *progress*

bar فيقوم بإظهار المعطيات بطريقة نسبية من الجزء الممتلئ "full" إلى الجزء الفارغ "empty"، مما يساعد المستخدم على الحصول على فكرة واضحة عن القيمة الموافقة لعنصر معين.

سنقوم في المثال التالي بإنشاء *Sldier* وشريط تقدم *progress bar* بحيث تتغير قيمة الشريط عندما يتم تحريك *Slider*:

```
//: Progress.java
// Using progress bars and sliders
package cl7.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.event.*;
import com.sun.java.swing.border.*;
public class Progress extends JPanel {
    JProgressBar pb = new JProgressBar();
    JSlider sb =
        new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
    public Progress() {
        setLayout(new GridLayout(2,1));
        add(pb);
        sb.setValue(0);
        sb.setPaintTicks(true);
        sb.setMajorTickSpacing(20);
        sb.setMinorTickSpacing(5);
        sb.setBorder(new TitledBorder("Slide Me"));
        sb.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                pb.setValue(sb.getValue());
            }
        });
        add(sb);
    }
    public static void main(String args[]) {
        Show.inFrame(new Progress(), 200, 150);
    }
}
```

```
} ///:~
```

كما تلاحظ فإن عملية إنشاء شريط التقدم *JProgressBar* بسيطة، أما لإنشاء *Jslider* فعليك تحديد الكثير من الخيارات كالاتجاه وعلامات القيمة الصغرى والكبرى.

الأشجار ...Trees

ويتم التعامل مع الأشجار من خلال الصف *JTree* حيث يمكنك إنشاء شجرة جديدة بشكل بسيط جداً، كأن تقول مثلاً:

```
add(new JTree(  
    new Object[] {"this", "that", "other"}));
```

وستولد هذه التعليمة شجرة أولية *primitive tree*.

وتعتبر الأشجار من أهم العناصر التي تحتويها مكتبة *Swing*. يوضح المثال التالي كيفية استخدام مكونات الشجرة الافتراضية *default tree*، التي تزودك بها هذه المكتبة، من أجل إظهار شجرة ضمن بريمج *applet*. وعندما تقوم بضغط زر، تتم إضافة شجرة فرعية تحت العقدة الحالية (وفي حال لم يتم اختيار عقدة، يتم استخدام عقدة الجذر):

```
//: Trees.java  
// Simple Swing tree example. Trees can be made  
// vastly more complex than this.  
package cl7.swing;  
import java.awt.*;  
import java.awt.event.*;  
import com.sun.java.swing.*;  
import com.sun.java.swing.tree.*;  
// Takes an array of Strings and makes the  
// first  
// element a node and the rest leaves:  
class Branch {  
    DefaultMutableTreeNode r;  
    public Branch(String[] data) {  
        r = new DefaultMutableTreeNode(data[0]);
```

```

        for(int i = 1; i < data.length; i++)
            r.add(new DefaultMutableTreeNode(data[i]));
    }
    public DefaultMutableTreeNode node() {
        return r;
    }
}

public class Trees extends JPanel {
    String[][] data = {
        { "Colors", "Red", "Blue", "Green" },
        { "Flavors", "Tart", "Sweet", "Bland" },
        { "Length", "Short", "Medium", "Long" },
        { "Volume", "High", "Medium", "Low" },
        { "Temperature", "High", "Medium", "Low" },
        { "Intensity", "High", "Medium", "Low" },
    };
    static int i = 0;
    DefaultMutableTreeNode root, child, chosen;
    JTree tree;
    DefaultTreeModel model;
    public Trees() {
        setLayout(new BorderLayout());
        root = new DefaultMutableTreeNode("root");
        tree = new JTree(root);
        // Add it and make it take care of scrolling:
        add(new JScrollPane(tree),
            BorderLayout.CENTER);
        // Capture the tree's model:
        model = (DefaultTreeModel) tree.getModel();
        JButton test = new JButton("Press me");
        test.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent
            e){
                if(i < data.length) {
                    child = new Branch(data[i++]).node();
                    // What's the last one you clicked?
                    chosen = (DefaultMutableTreeNode)

```

```

        tree.getLastSelectedPathComponent();
        if(chosen == null) chosen = root;
        // The model will create the
        // appropriate event. In response, the
        // tree will update itself:
        model.insertNodeInto(child, chosen, 0);
        // This puts the new node on the
        // currently chosen node.
    }
}
});
// Change the button's colors:
test.setBackground(Color.blue);
test.setForeground(Color.white);
JPanel p = new JPanel();
p.add(test);
add(p, BorderLayout.SOUTH);
}
public static void main(String args[]) {
    Show.inFrame(new Trees(), 200, 500);
}
} ///:~

```

الصف الأول Branch عبارة عن أداة تأخذ مصفوفة سلاسل محارف String، وتقوم ببناء عقد الشجرة DefaultMutableTreeNode، حيث تعتبر السلسلة الأولى جذر root الشجرة، أما بقية السلاسل فستكون أوراق leaves هذه الشجرة. يمكن بعد ذلك استدعاء الطريقة () node من أجل توليد جذر هذا الفرع.

أما الصف Trees فيحتوي على مصفوفة ثنائية الأبعاد من سلاسل المحارف String، والتي ستتكون من خلالها فروع branches هذه الشجرة. وتحتوي عناصر الصف DefaultMutableTreeNode على عقد الشجرة، أما التمثيل الفيزيائي لهذه الشجرة فسيتم التحكم به من خلال الصف JTree والنموذج model المرتبط به DefaultTreeModel.

لاحظ هنا أنه عندما تتم إضافة عنصر JTree إلى البريمج، سيتم تغليفه ضمن الصف JScrollPane والذي سيقوم بإجراء عملية الانزلاق التلقائية ضمن الشجرة.

وكما ذكرنا يتم التحكم بالصف *JTree* من خلال النموذج *model* المرتبط به. وعندما نقوم بإجراء أي تغيير على هذا النموذج، فسيقوم بتوليد حدث يطلب من العنصر *JTree* إجراء التعديلات الضرورية على التمثيل المرئي للشجرة. أما في الطريقة *init()*، فيتم التقاط النموذج باستدعاء *getModel()*. وعندما نقوم بالضغط على زر، يتم إنشاء فرع جديد، وسيظهر المكون الجديد فيه. أما الطريقة *insertNodeInto()* فسنقوم بإجراء جميع الأعمال اللازمة لتغيير الشجرة وإجراء مختلف التعديلات عليها.

الجداول Tables ...

تعتبر الجداول في مكتبة Swing من العناصر الفعالة والقوية جدا. ولقد كان الهدف الأساسي من إنشائها توليد واجهة شبكية لقواعد المعطيات من خلال (Java) JDBC (DataBase Connectivity) التي تمت مناقشتها في الفصل السابق. وتمكنك الجداول في Swing من التعامل مع أساسيات صفحات العمل spreadsheet وبإمكانها إنشاء جدول بسيط نسبيا JTable.

ويتم من خلال الصف JTable التحكم بكيفية إظهار المعطيات، أما الصف TableModel فتتحكم بالمعطيات نفسها. لذلك من أجل إنشاء عنصر JTable تحتاج أولا إلى إنشاء عنصر TableModel. ويمكنك تنفيذ الواجهة TableModel بشكل كامل، لكن يفضل التوريث من الصف AbstractTableModel:

```
//: Table.java
// Simple demonstration of JTable
package cl7.swing;
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.table.*;
import com.sun.java.swing.event.*;
// The TableModel controls all the data:
class DataModel extends AbstractTableModel {
    Object[][] data = {
        {"one", "two", "three", "four"},
        {"five", "six", "seven", "eight"},
        {"nine", "ten", "eleven", "twelve"},
    };
    // Prints data when table changes:
    class TML implements TableModelListener {
```

```

public void tableChanged(TableModelEvent e)
{
    for(int i = 0; i < data.length; i++) {
        for(int j = 0; j < data[0].length; j++)
            System.out.print(data[i][j] + " ");
        System.out.println();
    }
}

DataModel() {
    addTableModelListener(new TML());
}

public int getColumnCount() {
    return data[0].length;
}

public int getRowCount() {
    return data.length;
}

public Object getValueAt(int row, int col) {
    return data[row][col];
}

public void
setValueAt(Object val, int row, int col) {
    data[row][col] = val;
    // Indicate the change has happened:
    fireTableDataChanged();
}

public boolean
isCellEditable(int row, int col) {
    return true;
}
};

public class Table extends JPanel {
    public Table() {
        setLayout(new BorderLayout());
        JTable table = new JTable(new
            DataModel());
        JScrollPane scrollpane =

```

```

JTable.createScrollPaneForTable(table);
add(scrollpane, BorderLayout.CENTER);
}
public static void main(String args[]) {
    Show.inFrame(new Table(), 200, 200);
}
} ///:~

```

يحتوي عنصر *DataModel* على مصفوفة من المعطيات، و يمكنك الحصول على المعطيات من بعض المصادر الأخرى كقواعد المعطيات. ويقوم الباني باستخدام عنصر *TableModelListener* الذي يطبع المصفوفة في كل مرة يتم فيها تغيير الجدول. وتتبع بقية الطرق نفس التسميات الاصطلاحية للحبيبات *Beans*، وتستخدم من قبل *JTable* عندما ترغب بإظهار المعلومات في *DataModel*.

اللوحات المبوبة ...*Tabbed Panes*

رأينا في الفصول الماضية كيفية استخدام التخطيط *CardLayout*، وكيفية إدارة عملية القلب المزعة بين البطاقات. لحسن الحظ فلقد حلت مكتبة *Swing* جميع الصعوبات التي كنت تراها من قبل، حيث أنت بالصف *JTabbedPane* الذي يقوم بمعالجة جميع التبويبات *tabs* وعمليات القلب *switching* وغير ذلك. يوضح المثال التالي كيفية إنشاء اللوحات المبوبة:

```

//: Tabbed.java
// Using tabbed panes
package c17.swing;
import java.awt.*;
import com.sun.java.swing.*;
import com.sun.java.swing.border.*;
public class Tabbed extends JPanel {
    static Object[][] q = {
        { "Felix", Borders.class },
        { "The Professor", Buttons.class },
        { "Rock Bottom", ButtonGroups.class },
    }
}

```

```

{ "Theodore", Faces.class },
{ "Simon", Menus.class },
{ "Alvin", Popup.class },
{ "Tom", ListCombo.class },
{ "Bugs", Trees.class },
{ "Daffy", Table.class },
};
static JPanel makePanel(Class c) {
    String title = c.getName();
    title = title.substring(
        title.lastIndexOf('.') + 1);
    JPanel sp = null;
    try {
        sp = (JPanel)c.newInstance();
    } catch(Exception e) {
        System.out.println(e);
    }
    sp.setBorder(new TitledBorder(title));
    return sp;
}
public Tabbed() {
    setLayout(new BorderLayout());
    JTabbedPane tabbed = new JTabbedPane();
    for(int i = 0; i < q.length; i++)
        tabbed.addTab((String)q[i][0],
            makePanel((Class)q[i][1]));
    add(tabbed, BorderLayout.CENTER);
    tabbed.setSelectedIndex(q.length/2);
}
public static void main(String args[]) {
    Show.inFrame(new Tabbed(), 460, 350);
}
} ///:~

```

كما ترى ضمن الباني `Tabbed()`، توجد طريقتان هامتان مستخدمتان: الأولى `addTab()` لإنشاء تبويب جديد، والثانية `setSelectedIndex()` لاختيار لوحة البدء.

وعندما تقوم باستدعاء الطريقة `addTab()` يتم تزويدها بالسلسلة `String` الخاصة بالتبويب، إضافة إلى عنصر `Component`، والذي سيتم إظهاره في اللوحة. أما الطريقة `makePanel` فنقوم بأخذ عنصر `Class` إلى الصف الذي ترغب بإنشائه، ونستخدم `newInstance()` لإنشاء هذا الصف. ونقوم كذلك بإضافة عنصر `TitledBorder` الذي يحتوي على اسم الصف، ونقوم بإرجاع النتيجة كعنصر `JPanel` الذي سيستخدم في `addTab()`.

وعندما نقوم بتنفيذ هذا البرنامج ستجد بأن العنصر `JTabbedPane` يقوم وبشكل تلقائي بتكديس التبويبات `tabs` إذا وجد الكثير منها في نافذة وحيدة.



Thinking in Java, Bruce Eckel, Prentice Hall PTR, 1998.

Getting Started with Oracle AppBuilder for Java, Matthew Kagle, Oracle Press, 1998.

Java Unleashed, Second Edition, by Michael Morrison et al. 1996.

Hacking Java : The Java Professional's Resource Kit, Mark Wutka, Que, 1996.

Java Script Manual of Style, Ziff-Davis Press Development Group, ZDPRESS, 1996.

Java Unleashed, SAMS.NET Development Group, SAMS.NET, 1996.

Java Developer's Guide, Jamie Jaworski, SAMS.NET, 1996.

Developing Professional Java Applets, H.C. Hopson and Stephen E. Ingram, SAMS.NET, 1996.

Tricks of the Java Programming Gurus, Glenn Vanderburg, SAMS.NET, 1996.

Java Developer's reference, SAMS.NET, 1996.

Teach Yourself Java 1.1 Programming in 24 Hours, Rogers Cadenhead, SAMS.NET, 1996.

Presenting Java Beans, Michael Morrison, SAMS.NET, 1996.

Java 1.1 Unleashed, Third Edition, Bankston and Seifert, SAMS.NET, 1996.

مجلة Byte - الأعداد: ٨ (١٩٩٦)، ٢ (١٩٩٧)، ٣ (١٩٩٧).

مجلة المعلوماتي - الحاسوب والتقنيات، السنة السابعة/العدد الخامس والستون/آذار ١٩٩٨.





إصدار <i>Version</i>	إرساء <i>Install</i>	إبدال <i>Swapping</i>
إضافي <i>Additional</i>	إرسال <i>Transmission</i>	إتيان <i>Download</i>
إضمائة <i>Aggregation</i>	إزاحة نحو الخارج <i>Outdent</i>	إجراء <i>Procedure</i>
إطار عمل <i>Framework</i>	إزاحة نحو الداخل <i>Indent</i>	إجرائية <i>Process</i>
إعلام <i>Notification</i>	أساس <i>Radix</i>	إحالة <i>Transfer</i>
إعلام بالخروج <i>Sign out</i>	أساسي <i>Fundamental</i>	أخطوطة <i>Script</i>
إعلام بالدخول <i>Sign on</i>	أسلوب <i>Style</i>	أخطوطة <i>Script</i>
إقلاع <i>Booting</i>	إسناد <i>Assignment</i>	إخفاق <i>Failure</i>
إقلاع <i>Restart</i>	إشارة <i>Signal</i>	أداة <i>Tool</i>
آلة <i>Machine</i>	إشراف <i>Supervision</i>	إدارة <i>Administration</i>
إلغاء الأمر <i>Cancel</i>	إشعار <i>Acknowledgement</i>	إدارة <i>Management</i>
أمانة <i>Tag</i>	إصدار <i>Release</i>	إذاعة تعدنية <i>Multicasting</i>
أمر <i>Command</i>		إرساء <i>Install</i>

برمجيات <i>Software</i>	اختبار <i>Test</i>	أمر <i>Order</i>
برمجيات وسيطة <i>Middleware</i>	اختيار <i>OR</i>	آمن <i>save</i>
برنامج <i>Program</i>	اختيار مقصور <i>XOR</i>	أمن <i>Security</i>
بروتوكول <i>Protocol</i>	استخراج <i>Extraction</i>	أمين <i>Secure</i>
برمج <i>Applet</i>	استخلاص <i>Synthesis</i>	أنبوب <i>Pipe</i>
بشكل صريح <i>Explicitly</i>	استرداد <i>Recover</i>	أنبوب <i>Pipe</i>
بطاقة <i>Card</i>	استيقان <i>Authentication</i>	أنبوب <i>Tube</i>
بُعد <i>Dimention</i>	اشبك <i>Plug</i>	إنشاء <i>Creation</i>
بعيد <i>Remote</i>	افتراضياً <i>By Default</i>	أنصوية <i>Semaphore</i>
بند <i>Item</i>	اكتساب <i>Acquisition</i>	إنفاذ <i>Implementation</i>
بنيان <i>Architecture</i>	التحام <i>Junction</i>	إنقال <i>Relay</i>
بنية <i>Structure</i>	الوصول العشوائي <i>Random Access</i>	إنكفاء <i>Regression</i>
بوابة <i>Gate</i>	امتداد <i>Extension</i>	أولي <i>Primary</i>
بيان <i>Graph</i>	انتشار <i>Diffusion</i>	إيثاق <i>Bibding</i>
بيانيات <i>Graphics</i>	باع <i>Span</i>	ابتدائي <i>Initial</i>
تأخير <i>Delay</i>	باني <i>Constructor</i>	ابتدال <i>Switching</i>
تأصلي <i>Native</i>	بايت <i>Byte</i>	أبدأ <i>Start</i>
تأصلي (a.) <i>Native(a.)</i>	بت <i>Bit</i>	ابن <i>Child</i>
تبادل <i>Exchange</i>	بث <i>Diffusion</i>	اتحاد <i>Union</i>
تباعد <i>Spacing</i>	بث <i>Broadcasting</i>	اتصال <i>Communication</i>
تبذنة <i>Initialize</i>	بدء التشغيل <i>Start up</i>	اتصالات
تتالي <i>Cascade</i>	بذالة <i>Switch</i>	اتصالات <i>Telecommunications</i>
		احتواء <i>Encapsulation</i>



<i>Instruction</i> تعليمية	<i>Messaging</i> تراسل	<i>Consortium</i> تجمّع
<i>Ciphering</i> تعمية	<i>Upgrading</i> ترقية	<i>Aggregate</i> تجميع
<i>Encryption</i> تعمية	<i>Composition</i> تركيب	<i>Assembly</i> تجميع
<i>Feedback</i> تغذية راجعة	<i>Synthesis</i> تركيب	<i>Equipment</i> تجهيزات
<i>Wrap</i> تغليف	<i>Syntax</i> تركيب نحوي	<i>Alliance</i> تحالف
<i>Mapping</i> تقابل	<i>Coding</i> ترميز	<i>Packaged</i> تحزيم
<i>Exchange</i> تقايض	<i>Modulation</i> ترنيم	<i>Annotation</i> تحشية
<i>Emulation</i> تقليد	<i>Record</i> تسجيلة	<i>Allocation</i> تخصيص
<i>Secrecy</i> تكتم	<i>Serial</i> تسلسلي	<i>Acquisition</i> تحصيل
<i>Iteration</i> تكرار	<i>Routing</i> تسيير	<i>Check</i> تحقّق
<i>Analog</i> تماثلي	<i>Ciphering</i> تشفير	<i>Realization</i> تحقيق
<i>Format</i> تميمق	<i>Distortion</i> تشوّه	<i>Control</i> تحكّم
<i>Multiplexing</i> تضديد	<i>Application</i> تطبيق	<i>Overload</i> تحميل
<i>Implement</i> تنفيذ	<i>Concurrent</i> تعاقب	<i>Morphing</i> تحويل
<i>Execution</i> تنفيذ	<i>Manipulation</i> تعامل	<i>Transaction</i> تحويل
<i>Parallelism</i> تواز	<i>Treatment</i> تعامل	<i>Attribution</i> تخصيص
<i>Upcasting</i> توجيه للأعلى	<i>Expression</i> تعبير	<i>Layout</i> تخطيط
<i>Steganography</i> تورية	تعديّة الأشكال	<i>Planning</i> تخطيط
<i>Inheritance</i> توريث	<i>Polymorphism</i>	<i>Authorization</i> تخويل
<i>Expansion</i> توسيع	<i>Modulation</i> تعديل	<i>Flow</i> تدفّق
<i>Specification</i> توصيف	<i>Definition</i> تعريف	<i>Flux</i> تدفق
<i>Handling</i> تولّي	تعقّب رجوعي	<i>Auditing</i> تنقيق
<i>Generic</i> توليدي	<i>Backtracking</i>	<i>Notation</i> تدوين
	<i>Suspension</i> تعليق	

خطوط <i>Outline</i>	حجم <i>Size</i>	تومنة <i>Semaphore</i>
خلال <i>fault</i>	حجم <i>Volume</i>	ثانوي <i>Secondary</i>
دائم <i>Permanent</i>	حد <i>Limit</i>	ثمانية <i>Octet</i>
دارئ <i>Buffer</i>	حدث <i>Event</i>	جدول <i>Table</i>
دالة <i>Function</i>	حذف <i>Delete</i>	جدولة <i>Scheduler</i>
دفعي <i>Batch</i>	حزمة <i>Package</i>	جدولة <i>Schedule</i>
دقق <i>Stream</i>	حزمة <i>Band</i>	جذر <i>Root</i>
دقق <i>Stream</i>	حزمة <i>Package</i>	جذر الأساس <i>Radical</i>
دقة <i>Accurate</i>	حفظ <i>Save</i>	جسر <i>Bridge</i>
دقة <i>Precision</i>	حلف <i>Alliance</i>	جك <i>Jack</i>
دلالة <i>Semantics</i>	حلقة <i>Loop</i>	جمع (ة) <i>Collection</i>
دليل <i>Directory</i>	حلقة <i>Ring</i>	جمعية <i>Society</i>
دورة <i>Cycle</i>	حمل مضاف <i>Overhead</i>	جملة <i>Suite</i>
ذاكرة رام <i>RAM</i>	حيز <i>Slot</i>	جهاز <i>Device</i>
ذاكرة رام ديناميكية <i>DRAM</i>	خازنة <i>Repository</i>	جودة <i>Quality</i>
ذاكرة رام سكونية <i>SRAM</i>	خاص <i>Particular</i>	جوهري <i>Essential</i>
ذاكرة روم <i>ROM</i>	خاص <i>Special</i>	حاشية <i>Footnote</i>
ذاكرة روم قابلة للبرمجة <i>PROM</i>	خاصة <i>Property</i>	حافة <i>Border</i>
ذاكرة روم قابلة للمحو والبرمجة <i>EPROM</i>	خاصية <i>Attribute</i>	حافظة <i>Cartridge</i>
ذاكرة قراءة فقط <i>ROM</i>	خصوصي <i>Private</i>	حالة <i>State</i>
ذاكرة وصول عشوائي <i>RAM</i>	خصوصية <i>Privacy</i>	حامل <i>Carrier</i>
	خط توارد <i>Pipeline</i>	حامل <i>Chassis</i>
	خطأ <i>Error</i>	حجز <i>Reservation</i>



Slot شق	Privacy سرية	View رؤية
Figure شكل	Join سضم	Major رئيس
Plain صرف	Surface سطح	Master رئيس
Class صف	Desktop سطح المكتب	Main رئيسي
Attribute صفة	Static سكوني	Association رابطة
Quality صفة	Safety سلامة	Quartet رباعية
Valve صمام	Chain سلسلة	Nibble رباعية
Tube صمام	Series سلسلة	Quadbit رباعية بتات
Voice صوت	String سلسلة محارف	Connection ربط
Sound صوت	Audio سمعي	Packet رزمة
Acoustic صوتي	Driver سواق	Packet رزمة
Image صورة	Drive سواقة	Observation رصد
Picture صورة	Master سيد	Monitoring رقابة
Formulation صياغة	Silicon سيليكون	Chip رقاقة
Formula صيغة	Sign شارة	Digit رقم
Precision ضبط	Screen شاشة	Symbol رمز
Opposite ضد	Global شامل	Numeral رمز الرقم
AND ضم	Personal شخصي	Button زر
Optical ضوئي	Wafer شريحة	Group زمرة
Map(n) طباق	Condition شرط	Turnaround زمن دورة
Layer طبقة	Enterprise شركة	time
Tier طبقة	Company شركة	Register سجل
Peripheral طرفية	Slice شريحة	Secrecy سرية

<i>Mistake</i> غلط	<i>Token</i> علام	<i>Method</i> طريقة
<i>Underflow</i> غيض	<i>Tag</i> علامة	<i>Approach</i> طريقة
<i>Category</i> فئة	<i>Token</i> علامة	<i>Method</i> طريقة
<i>Space</i> فراغ	<i>Mark</i> علامة	<i>Kit</i> طقم
<i>Corruption</i> فساد	<i>Tag</i> علامة	<i>Suite</i> طقم
<i>Space</i> فضاء	<i>Prompt</i> علامة تقبل	<i>Request</i> طلب
<i>Action</i> فعل	<i>Bug</i> علة	<i>Log Off</i> طلب الخروج
<i>Backbone</i> فقار	<i>Cryptology</i> علم التعمية	<i>Logout</i> طلب الخروج
<i>Index</i> فهرس	<i>On-Line</i> على الخط	<i>Log On</i> طلب الدخول
<i>Physical</i> فيزيائي	<i>Lifetime</i> عمر	<i>Login</i> طلب الدخول
<i>Overflow</i> فيض	<i>Job</i> عمل	<i>Core</i> طوق
<i>Menu</i> قائمة	<i>Operation</i> عملية	<i>Universal</i> عالمي
<i>Legend</i> قائمة تفسيرية	<i>Public</i> عمومي	<i>General</i> عام
<i>Plug</i> قابس	<i>Agent</i> عميل	<i>General</i> عام
<i>Coupler</i> قارن	<i>Object</i> عنصر	<i>Factor</i> عامل
<i>Template</i> قالب	<i>Element</i> عنصر	<i>Statement</i> عبارة
<i>Lexicon</i> قاموس	<i>Object</i> عنصر	<i>Gateway</i> عبارة
<i>Shell</i> قشرة	<i>Cluster</i> عنقود	<i>Hardware</i> عتاديات
<i>Porpose</i> قصد	<i>Label</i> عنوان	<i>Number</i> عدد
<i>Sector</i> قطاع	<i>Recursive</i> عودي	<i>Wide</i> عريض
<i>Lock</i> قفل	<i>Recursion</i> عودية	<i>Default</i> عطل
<i>Canal</i> قناة	<i>Purpose</i> غاية	<i>Failure</i> عطل
<i>Channel</i> قناة	<i>Object</i> غرض	<i>Fault</i> عطل



<i>Simultaneous</i> متواقت	<i>Establishment</i> مؤسسة	<i>Size</i> قياس
<i>Synchronous</i> متواقت	<i>Handle</i> مؤشر	<i>Standard</i> قياسي
<i>Example</i> مثال	<i>Cursor</i> مؤشر	<i>Paradigm</i> قيس
<i>Domain</i> مجال	<i>Pointer</i> مؤشر	<i>Cable</i> كبل
<i>Interval</i> مجال	<i>Secured</i> مأمون	<i>Block</i> كتلة
<i>Range</i> مجال	<i>Interchange</i> مبادلة	<i>Card</i> كرت
<i>Abstract</i> مجرد	<i>Transaction</i> مبادلة	<i>Glossary</i> كشف
<i>Folder</i> مجلد	<i>Syntax</i> مبنى	<i>Word</i> كلمة
<i>Volume</i> مجلد	<i>Sequential</i> متتال	<i>Heap</i> كومة
<i>Blocked</i> مجمد	<i>Sequence</i> متتالية	<i>Heap</i> كومة
<i>Assembler</i> مجمع	<i>Variable</i> متحول	<i>Entity</i> كيان
<i>Collector</i> مجمع	<i>Compact</i> متراص	<i>List</i> لائحة
<i>Garbage</i> مجمع النفايات	<i>Compiler</i> مترجم	<i>Table</i> لائحة
<i>Collector</i>	<i>Synchronous</i> متزامن	<i>Profile</i> لائحة
<i>Group</i> مجموعة	<i>Serial</i> متسلسل	<i>Banner</i> لافتة
<i>Set</i> مجموعة	<i>On-Line</i> متصل	<i>Core</i> لب
<i>Alignment</i> محاذاة	<i>Embedded</i> متضمن	<i>Label</i> لصاقة
<i>Simulation</i> محاكاة	<i>Related</i> متعلق بـ	<i>Palette</i> لوحة
<i>Identifier</i> محدد	<i>Variable</i> متغير	<i>Panel</i> لوحة
<i>Argument</i> محدد	<i>Integrated</i> متكامل	<i>Board</i> لوحة، بطاقة، لوح
<i>Specific</i> محدد	<i>Contiguous</i> متلاصق	<i>Tablet</i> لوحة
<i>Character</i> محرف	<i>Parallel</i> متواز	<i>Operand</i> مؤثر في
<i>Engine</i> محرك	<i>Compatible</i> متوافق	<i>Outlet</i> مأخذ

<i>Phreak</i> مسترق	<i>Built in</i> مدموج	<i>Motor</i> محرك
<i>User</i> مستعمل	<i>Extent</i> مدى	<i>Trajectory</i> محرك
<i>Stationary</i> مستقر	<i>Scope</i> مدى	<i>Packaged</i> محزوم
<i>Persistent</i> مستمر	<i>Administrator</i> مدير	<i>Sensor</i> محس
<i>Warehouse</i> مستودع	<i>Manager</i> مدير	<i>Plain</i> محض
<i>Mart</i> مستودع محلي	<i>Observation</i> مراقبة	<i>Console</i> محكم
<i>Level</i> مستوى	<i>Connector</i> مرتبط	<i>Location</i> محل
<i>Listing</i> مسردة	<i>Grade</i> مرتبة	<i>Portable</i> محمول
<i>Bus</i> مسرى	<i>Reference</i> مرجع	<i>Transducer</i> محوال
<i>Track</i> مسلك	<i>Reflector</i> مرداد	<i>Cracker</i> مخترق
<i>Router</i> مسير	<i>Repeater</i> مردد	<i>Specific</i> مختص
<i>Operator</i> مشغل	<i>Packed</i> مرزوم	<i>Storage</i> مخزن
<i>View</i> مشهد	<i>Attachment</i> مرفقة	<i>Store</i> مخزن
<i>Indicator</i> مشيرة	<i>Monitor</i> مراقب	<i>Storefront</i> مخزن عرض
<i>Resource</i> مصدر	<i>Compound</i> مركب	<i>Diagram</i> مخطط
<i>Source</i> مصدر	<i>Composite</i> مركب	<i>Schema</i> مخطط
<i>Folder</i> مصنف	<i>Concentrator</i> مركزة	<i>Chart</i> مخطط بياني
<i>Add-On</i> مضاف	<i>Duplex</i> مزدوج	مخططات هندسية
<i>Embedded</i> مضمن	<i>Path</i> مسار	<i>Schematic</i>
<i>Host</i> مضيف	<i>Distance</i> مسافة	<i>Orbit</i> مدار
<i>Terminal</i> مطراف	<i>Routine</i> مساق	<i>Period</i> مدة
<i>Enclosed</i> مطوق	<i>Auxiliary</i> مساند	<i>Incorporated</i> مدمج
<i>Piggyback</i> مطوي	<i>User</i> مستخدم	<i>Destructor</i> مدمر



منفصل عن الخط - <i>Off-Line</i>	مقطع جانبي <i>Profile</i>	مظهر <i>Display</i>
منقلة <i>Relay</i>	مكتب <i>Office</i>	معالجة <i>Processing</i>
منقول <i>Mobile</i>	مكدس <i>Stack</i>	معامل <i>Operator</i>
منه <i>Terminator</i>	مكدس <i>Stack</i>	معامل <i>Coefficient</i>
مهلة <i>Delay</i>	مكنز <i>Thesaurus</i>	معاملة <i>Transaction</i>
مهلة <i>Time out</i>	مكون (ة) <i>Component</i>	معاملة <i>Treatment</i>
مهمة <i>Task</i>	ملتحقة <i>Widget</i>	معاود <i>Recurrent</i>
موائمة <i>Adapter</i>	ملحقات <i>Accessories</i>	معجم <i>Dictionary</i>
مواردة <i>Pipelining</i>	ملف <i>File</i>	معدل <i>Rate</i>
مواصفة <i>Specification</i>	ممر <i>Passage</i>	معدل التدفق <i>Throughput</i>
موحد <i>Uniform</i>	من بعد <i>Remote</i>	معنى <i>Semantics</i>
مرصل <i>Conductor</i>	ملنس <i>Lurk</i>	معيار <i>Criteria</i>
موضع <i>Position</i>	منسق <i>Module</i>	معيار <i>Standard</i>
موقع <i>Site</i>	منشأ <i>Origin</i>	معياري <i>Standard</i>
موقف <i>Situation</i>	منصب <i>Rack</i>	معييرة <i>Standardization</i>
ميزة <i>Characteristic</i>	منضاد <i>Multiplexer</i>	مغرز <i>Pin</i>
ميزة <i>Feature</i>	منطقة <i>Area</i>	مفرد <i>Simplex</i>
ميفاق <i>Protocol</i>	منطقة <i>Region</i>	مفردات <i>Vocabulary</i>
ناطر [الطباعة] <i>Spooler</i>	منطقة <i>Zone</i>	مفردة <i>Item</i>
ناقل <i>Carrier</i>	منظر <i>Landscape</i>	مفسر <i>Interpreter</i>
ناقل <i>Conductor</i>	منظمة <i>Organization</i>	مقبس <i>Socket</i>
نبلة <i>Nibble</i>	منفذ <i>Port</i>	مقطع <i>Section</i>
		مقطع <i>Segment</i>

نحو <i>Syntax</i>	نمط طابعي <i>Stereotype</i>	وصلة <i>Link</i>
نسخة <i>Copy</i>	نموذج <i>Form</i>	وصول <i>Access</i>
نسخة <i>Version</i>	نموذج <i>Model</i>	وضع <i>Status</i>
نسيب <i>Thread</i>	نموذج أولي <i>Prototype</i>	وكالة <i>Agency</i>
نطاق <i>Domain</i>	نهائي <i>Deadline</i>	وكيل <i>Agent</i>
نطاق <i>Extent</i>	نهاية <i>End</i>	يأتي <i>Download</i>
نطاق عمل <i>scope</i>	نواة <i>Kernal</i>	يؤشر <i>Point</i>
نفي <i>NOT</i>	نوع <i>Type</i>	يبحث <i>Research</i>
نفي اختيار <i>NOR</i>	نوعي <i>Generic</i>	يبحث <i>Search</i>
نفي اختيار مقصور <i>NXOR</i>	نوعية <i>Quality</i>	يبدأ <i>Start</i>
نفي ضم <i>NAND</i>	نيسب <i>Thread</i>	يبدئ <i>Initialize</i>
نقال <i>Mobile</i>	نيسبة <i>Threading</i>	يبدل <i>Convert</i>
نقر <i>Click</i>	واجهة <i>Interface</i>	يبطال <i>Override</i>
نقش <i>Pattern</i>	واسع <i>Wide</i>	يتجاوز <i>Overrun</i>
نقطة <i>Dot</i>	واصلة <i>Jumper</i>	يتخطى <i>Bypass</i>
نقطة <i>Point</i>	وثيقة <i>Document</i>	يترجم <i>Translate</i>
نقطة تفرع <i>Tap</i>	وحدة <i>Unit</i>	يتقاصر <i>Underrun</i>
نقطة وصل <i>Tap</i>	وحدة مستقلة <i>Entity</i>	يجدول <i>Tabulate</i>
نقل <i>Transfer</i>	وحدة نمطية <i>Module</i>	يجدول <i>Schedule</i>
نقل <i>Transmission</i>	وسيط <i>Parameter</i>	يجيز <i>Validate</i>
نقل <i>Transport</i>	وصف <i>Description</i>	يحدث <i>Update</i>
نمط <i>Mode</i>	وصل <i>Connection</i>	يحفظ <i>save</i>
نمط <i>Type</i>	وصلة <i>Junction</i>	يحمل <i>Port</i>



يحمل <i>Load</i>	يستنظم <i>Normalize</i>	يكبس <i>Press</i>
يحول <i>Convert</i>	يستهل <i>Initiate</i>	يلحق <i>Append</i>
يحول <i>Transform</i>	يسحب <i>Pull</i>	يمد <i>Extend</i>
يخزن <i>Store</i>	يسحب <i>Translate</i>	يمحي <i>Remove</i>
يدفع <i>Push</i>	يسقط <i>Suppress</i>	يمر <i>Forward</i>
يدمج <i>Merge</i>	يثشد <i>Tie</i>	يمر <i>Pass</i>
يربط <i>Associate</i>	يشعب <i>Fork</i>	يمرر <i>Pass</i>
يرتد <i>Revert</i>	يصير <i>Render</i>	يمعير <i>Standardize</i>
يرجع <i>Reference</i>	يضغط <i>Compress</i>	يناسب <i>Suit</i>
يرحل <i>Upload</i>	يضغط <i>Press</i>	ينشر <i>Propagate</i>
يرسل <i>Send</i>	يطابق <i>Match</i>	ينشر <i>Spread</i>
يرسل <i>Transmit</i>	يطلب <i>Request</i>	ينتهي <i>Terminate</i>
يرسل بالبريد <i>Post</i>	يعدل <i>Update</i>	ينزع <i>Pop</i>
يرسي <i>Install</i>	يعكس <i>Invert</i>	ينشئ <i>Originate</i>
يرقي <i>Upgrade</i>	يعكس <i>Reverse</i>	ينشر <i>Propagate</i>
يزيل <i>Remove</i>	يعيد <i>Return</i>	ينشر <i>Spread</i>
يزيل <i>Suppress</i>	يعيد الإقلاع <i>Reboot</i>	ينصب <i>Mount</i>
يستبعد <i>Suppress</i>	يغير <i>Transform</i>	ينظر <i>Spool</i>
يستحضر <i>Retrieve</i>	يقابل <i>Map(v)</i>	ينهي <i>Terminate</i>
يستدعي، ينادي <i>Call</i>	يقر <i>Validate</i>	يهيمن/يطغى <i>Override</i>
يسترجع <i>Restore</i>	يقلب <i>Invert</i>	يوسع <i>Expand</i>
يستفتح <i>Reset</i>	يقلب <i>Reverse</i>	

عناوين صدرت في سلسلة الرضا للمعلومات

اسم الكتاب	المؤلف	تاريخ النشر
١- بيئة النوافذ WINDOWS 3.11	م. أحمد شريك	١٩٩٤
٢- مبادئ الصيانة والشبكات	م. عبد الله أحمد	١٩٩٤
٣- معالجة النصوص MS WORD 6.0	د. هيثم البيطار	١٩٩٥
٤- ادخل إلى عالم WINDOWS 95	م. مهيب النكري	١٩٩٦
٥- قواعد البيانات MS ACCESS	زياد كمرجي - بيداء الزير	١٩٩٧
٦- توابع وماكروا في MS EXCEL 97	أ. زياد كمرجي	١٩٩٧
٧- مرجع تعليمي شامل لبرنامج معالجة النصوص MS WORD 97	د. هيثم البيطار	١٩٩٧
٨- مرجع تعليمي شامل في MS EXCEL 97	أ. زياد كمرجي	١٩٩٧
٩- مرجع تعليمي شامل في صيانة الحواسيب الشخصية	م. عبد الله أحمد	١٩٩٨
١٠- مرجع تعليمي في برنامج الرسم والتصميم الهندسي AUTOCAD 14	م. احسان مردود	١٩٩٨
١١- المرجع التدريبي الشامل لـ WINDOWS 98	م. إياد زوكار	١٩٩٨
١٢- ادخل إلى عالم WINDOWS 98	م. مهيب فواز النكري	١٩٩٨
١٣- الإنترنت وإنترانيت وتصميم المواقع	م. عبد الله أحمد	١٩٩٨
١٤- تكنولوجيا المعلومات	هاني شحادة الخوري	١٩٩٨
على أعتاب القرن الحادي والعشرين	د. يونس حيدر	١٩٩٩
١٥- الإدارة الاستراتيجية للشركات والمؤسسات	م. محمد حسن - م. بسام عزام	١٩٩٩
١٦- نظام الـ ISO 9004-1		

- ١٧- القائد المفكر حافظ الأسد
والمشروع التنموي الحضاري د.رياض عواد-أ.هاني الخوري ١٩٩٩
- ١٨- فن إدارة البشر د. محمد مرعي مرعي ١٩٩٩
- ١٩- المرجع الشامل لتعليمات برنامج AUTOCAD م. احسان المردود -م. وهبي معاد ١٩٩٩
- ٢٠- الدعاية والتسويق ومعاملة الزبائن م. حنا بللوز ١٩٩٩
- ٢١- المعلومات (المعلوماتية) ظروفها وآثارها الاقتصادية - الاجتماعية د. معن النكري ١٩٩٩
- ٢٢- المرجع الشامل لبرنامج 3D STUDIO MAX م. جورج عطا لله بركات ١٩٩٩
- ٢٣- دليل الجودة في المؤسسات والشركات د. طلال عبود-أ.ماهر العجي ١٩٩٩
- ٢٤-المرجع المفيد في علم شبكات الحواسيب د.معتصم شفا عمري ١٩٩٩
- ٢٥- ادخل إلى عالم ORACLE 8 م. مهيب النكري ١٩٩٩
- ٢٦- أسس إدارة الموارد البشرية د. محمد مرعي مرعي ١٩٩٩
- ٢٧- تعلم برنامج إدارة قواعد البيانات أ. زياد كمرجي - م. مهيب النكري ١٩٩٩
- ٢٨- الدليل الشامل لأساسيات الحاسوب والمعلوماتية م. عبد الله أحمد ١٩٩٩
- ٢٩- الكذبات العشر للعولة د. عدنان سليمان ١٩٩٩
- ٣٠- بعض مسائل الاقتصاد اللاسياسي د. مطانيوس حبيب ١٩٩٩
- ٣١- دليل إعادة تنظيم المؤسسات د. محمد مرعي مرعي ١٩٩٩
- ٣٢- الدراسات التسويقية ونظم معلومات التسويق د. طلال عبود - د. حسين علي ١٩٩٩
- ٣٣- مدخل إلى المعلوماتية الطبية م. جورج بركات - أ. هاني الخوري ١٩٩٩
- ٣٤- الدعاية والتسويق وفن التعامل مع الزبائن - جزء ٢ م. حنا بللوز ١٩٩٩

عناوين ستصدر قريباً

اسم الكتاب	المؤلف	تاريخ النشر المتوقع
١- العمل السكرتاري وبرنامج OUTLOOK ببداء الزير		١٩٩٩
٢- نظام الشبكات WINDOWS NT	م.عبد الله أحمد	١٩٩٩
٣-تصميم المواقع WEB DESIGN	م.عبد الله أحمد	١٩٩٩
٤- التسويق وإدارة الأعمال التجارية	م. إياد زوكار	١٩٩٩
٥-أمثلة وحالات عملية في EXCEL	م. إياد زوكار- م. نهال زركلي	١٩٩٩
٦-المعلوماتية الطبية	د.نبيل دك الباب	١٩٩٩
٧- مفاهيم حديثة في الإدارة المالية	د.دريد درغام	٢٠٠٠
٨- البرمجة في ACCESS BASIC	د.باسل الخطيب	٢٠٠٠
٩- أوراق ٨ - الجزء الثاني	م. مهيب النقري	٢٠٠٠
١٠- برنامج 3D MAX الجزء الثاني	م.جورج بركات	٢٠٠٠
١١- المرجع الأساسي للمعلوماتية	أ. شادي سيدا	٢٠٠٠



APPLETS

FRONT
PAGE



web

About as

Profile

JAVA SCRIPT

لغة JAVA، تخدم تطبيقات إنترنت وتصميم المواقع

× JAVA، لغة برمجة أصبحت منصة برمجية واسعة الانتشار .

× بدايتها كانت لوضع صفحات على "الوب" وهي اليوم منصة مستقلة

تماما عن أنظمة التشغيل .

× حمل سمات لغة " C++ " وتجاوزها بفوائد كامنة عديدة .

× حُتل جافا الآن مكاناً مهماً في أنظمة التشغيل الشبكية وبرمجيات قواعد البيانات .



سلسلة الرضا للمعلوم
دار الرضا للنشر